Java Collections Notes

Collection Hierarchy in Java | Collection Interface

In this tutorial, we will know another important topic **collection hierarchy in java** and collection interface.
In the previous tutorial, we have covered the basic points of the collections framework in Java with realtime examples, why do we need for collections framework if arrays are already available to store a group of objects.
If you have not familiar with all these concepts, I will recommend you to glance once.

Whenever you will go for any java technical interview, an interviewer can ask you one question related to Java collection hierarchy diagram. So, let's understand the basics.

Collection Hierarchy in Java

The hierarchy of the entire collection framework consists of four core interfaces such as Collection, List, Set, Map, and two specialized interfaces named SortedSet and SortedMap for sorting.

All the interfaces and classes for the collection framework are located in java.util package. The diagram of Java collection hierarchy is shown in the below figure.

Java Collections Notes

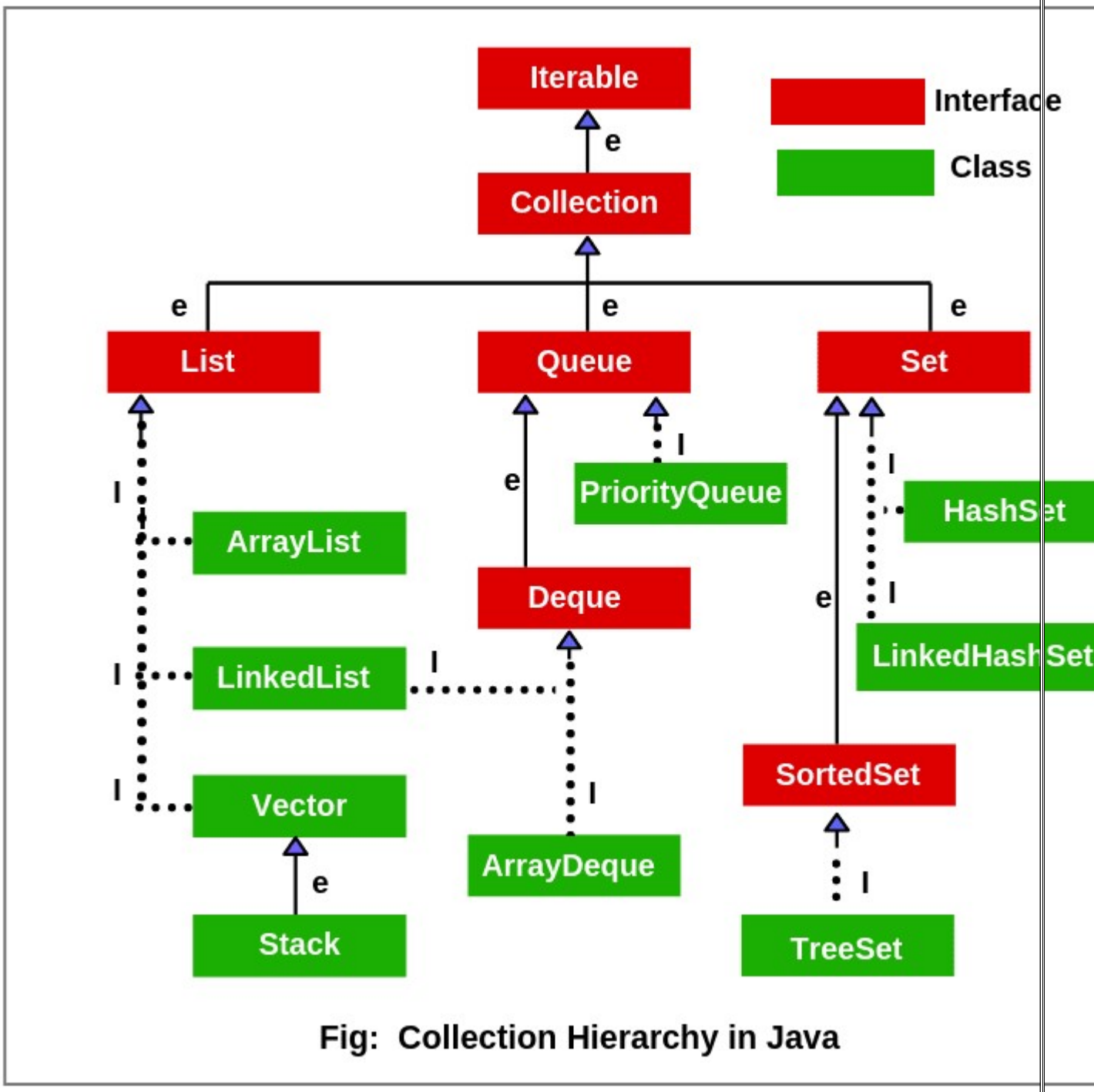


Fig: Collection Hierarchy in Java

e→ extends, I→ implements

Extends: Extends is a keyword that is used for developing inheritance between two classes and two interfaces.

Implements: Implements is a keyword used for developing inheritance between class and interface.

Java Collections Notes

Collection Interface in Java

➲ The basic interface of the collections framework is the Collection interface which is the root interface of all collections in the API (Application programming interface).

It is placed at the top of the collection hierarchy in java. It provides the basic operations for adding and removing elements in the collection.

➲ The Collection interface extends the Iterable interface. The iterable interface has only one method called iterator(). The function of the iterator method is to return the iterator object. Using this iterator object, we can iterate over the elements of the collection.
➲ List, Queue, and Set have three component which extends the Collection interface. A map is not inherited by Collection interface.

List Interface

➲ This interface represents a collection of elements whose elements are arranged sequentially ordered.

➲ List maintains an order of elements means the order is retained in which we add elements, and the same sequence we will get while retrieving elements.

➲ We can insert elements into the list at any location. The list allows storing duplicate elements in Java.

➲ ArrayList, vector, and LinkedList are three concrete subclasses that implement the list interface.

Set Interface

➲ This interface represents a collection of elements that contains unique elements. i.e, It is used to store the collection of unique elements.

➲ Set interface does not maintain any order while storing elements and while retrieving, we may not get the same order as we put elements. All the elements in a set can be in any order.

➲ Set does not allow any duplicate elements.

Java Collections Notes

➲ HashSet, LinkedHashSet, TreeSet classes implements the set interface and sorted interface extends a set interface.
➲ It can be iterated by using Iterator but cannot be iterated using ListIterator.

SortedSet Interface

➲ This interface extends a set whose iterator transverse its elements according to their natural ordering.

➲ TreeSet implements the sorted interface.

Queue Interface

➲ A queue is an ordered of the homogeneous group of elements in which new elements are added at one end(rear) and elements are removed from the other end(front). Just like a queue in a supermarket or any shop.

➲ This interface represents a special type of list whose elements are removed only from the head.

➲ LinkedList, Priority queue, ArrayQueue, Priority Blocking Queue, and Linked Blocking Queue are the concrete subclasses that implement the queue interface.

Deque Interface

➲ A deque (double-ended queue) is a sub-interface of queue interface. It is usually pronounced “deck”.

➲ This interface was added to the collection framework in Java SE 6.

➲ Deque interface extends the queue interface and uses its method to implement deque. The hierarchy of the deque interface is shown in the below figure.

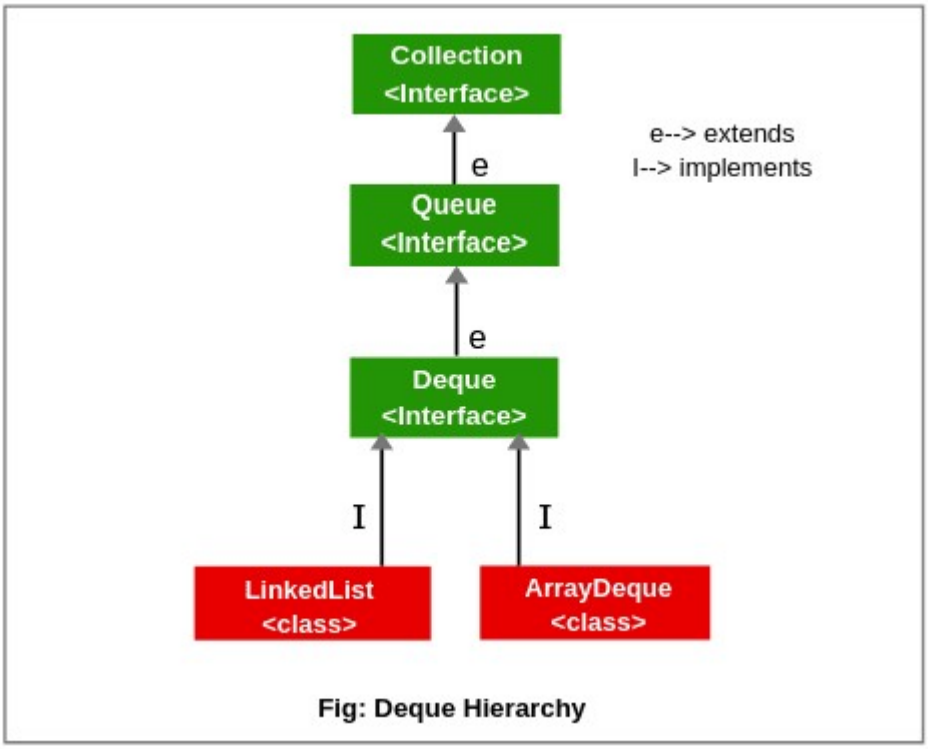


Fig: Deque Hierarchy

➲ It is a linear collection of elements in which elements can be inserted and removed from either end. i.e, it supports insertion and removal at both ends of an object of a class that implements it.

➲ LinkedList and ArrayDeque classes implement the Deque interface.

Map Interface

➲ Map interface is not inherited by the collection interface. It represents an object that stores and retrieves elements in the form of a Key/Value pairs and their location within the Map are determined by a Key.

Java Collections Notes

The hierarchy of the map interface is shown in the below figure.

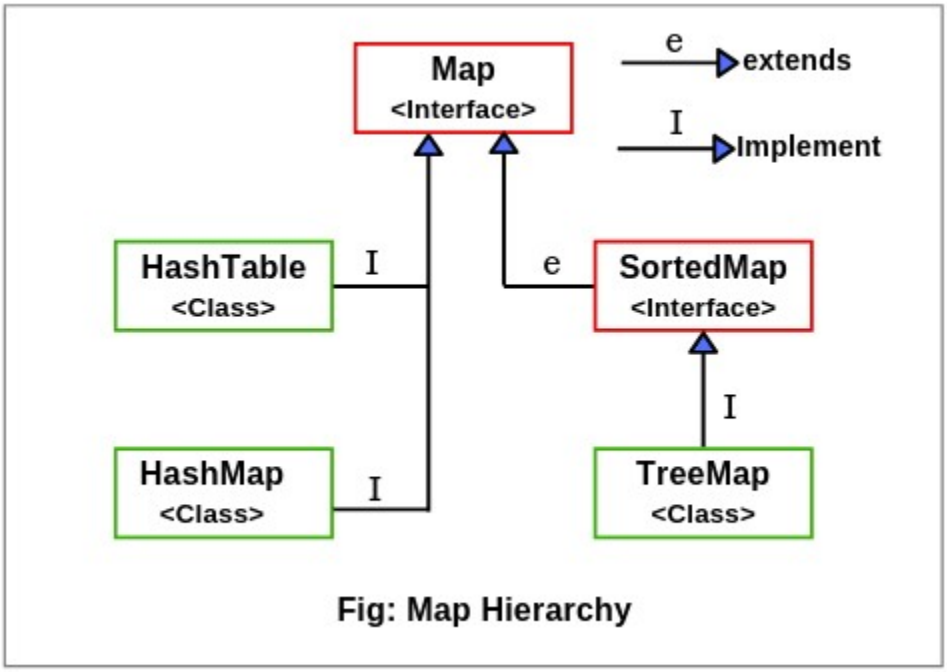


Fig: Map Hierarchy

➲ Map uses a hashing technique for storing key-value pairs.

➲ It doesn't allow to store the duplicate keys but duplicate values are allowed.

➲ HashMap, HashTable, LinkedHashMap, TreeMap classes implements Map interface.

SortedMap Interface

➲ This interface represents a Map whose elements are stored in their natural ordering. It extends the Map interface which in turn is implemented by TreeMap classes.

Methods of Collection Interface in Java

The Collection interface consists of a total of fifteen methods for manipulating elements in the collection. They are as follows:

Java Collections Notes

1. **add():** This method is used to add or insert an element in the collection. The general syntax for add() method is as follow:

```
add(Object element) : boolean
```

If the element is added to the collection, it will return true otherwise false, if the element is already present and the collection doesn't allow duplicates.

2. **addAll():** This method adds a collection of elements to the collection. It returns true if the elements are added otherwise returns false. The general syntax for this method is as follows:

```
addAll(Collection c) : boolean
```

3. **clear():** This method clears or removes all the elements from the collection. The general form of this method is as follows:

```
clear() : void
```

This method returns nothing.

4. **contains():** It checks that element is present or not in a collection. That is it is used to search an element. The general for contains() method is as follows:

```
contains(Object element) : boolean
```

This method returns true if the element is present otherwise false.

5. **containsAll():** This method checks that specified a collection of elements are present or not. It returns true if the calling collection contains all specified elements otherwise return false. The general syntax is as follows:

```
containsAll(Collection c) : boolean
```

6. **equals():** It checks for equality with another object. The general form is as follows:

```
equals(Object element) : boolean
```

7. **hashCode():** It returns the hash code number for the collection. Its return type is an integer. The general form for this method is:

```
hashCode() : int
```

8. **isEmpty():** It returns true if a collection is empty. That is, this method returns true if the collection contains no elements.

```
isEmpty() : boolean
```

9. **iterator():** It returns an iterator. The general form is given below:

```
iterator() : Iterator
```

10. **remove():** It removes a specified element from the collection. The general syntax is given below:

```
remove(Object element) : boolean
```

This method returns true if the element was removed. Otherwise, it returns false.

11. **removeAll():** The removeAll() method removes all elements from the collection. It returns true if all elements are removed otherwise returns false.

```
removeAll(Collection c) : boolean
```

12. **retainAll():** This method is used to remove all elements from the collection except the specified collection. It returns true if all the elements are removed otherwise returns false.

```
retainAll(Collection c) : boolean
```

13. **size():** The size() method returns the total number of elements in the collection. Its return type is an integer. The general syntax is given below:

```
size() : int
```

14. **toArray():** It returns the elements of a collection in the form of an array. The array elements are copies of the collection elements.

```
toArray() : Object[]
```

15. **Object[ ] toArray():** Returns an array that contains all the elements stored in the invoking collection. The array elements are the copies of the collection elements.

```
toArray(Object array[]) : Object[]
```

Collections Class in Java

The collections classes implement the collection interfaces. They are defined in java.util package. Some of the classes provide full implementations that can be used as it is.

Others are abstract that provide basic implementations that can be used to create concrete collections. A brief overview of each concrete collection class is given below.

1. **AbstractCollection:** It implements most of the collection interface. It is a superclass for all of the concrete collection classes.
2. **AbstractList:** It extends AbstractCollection and implements most of the List interface.
3. **AbstractQueue:** It extends AbstractCollection and implements the queue interface.
4. **AbstractSequentialList:** It extends AbstractList and uses sequential order to access elements.
5. **AbstractSet:** Extends AbstractCollection and implements most of the set interface.
6. **ArrayList:** It implements a dynamic array by extending AbstractList.
7. **EnumSet:** Extends AbstractSet for use with enum elements.
8. **HashSet:** Extends AbstractSet for use with a hash table.
9. **LinkedHashSet:** Extends HashSet to allow insertion-order iterations.
10. **LinkedList:** Implements a linked list by extending AbstractSequentialList.
11. **PriorityQueue:** Extends AbstractQueue to support a priority-based queue.

12. **TreeSet:** Extends AbstractSet and implements the SortedSet interface.

Java List Interface | Methods, Example

List in Java

➲ A **list in Java** is a collection for storing elements in sequential order. Sequential order means the first element, followed by the second element, followed by the third element, and so on.
A good realtime example of a list is a line of train bogies on a track: To get to the fourth bogie from the first bogie, we have to go through the second and third bogies in that order.

➲ Java list is a sub-interface of the collection interface that is available in java.util package. Sub interface means an interface that extends another interface is called sub interface. Here, the list interface extends the collection interface.

The general declaration of list interface in java is as follows:

```
public interface List<E> extends Collection<E>
```

➲ It is an ordered collection where elements are maintained by index positions because it uses an index-based structure. In the list interface, the order is retained in which we add elements. We will also get the same sequence while retrieving elements.

➲ It is used to store a collection of elements where duplicate elements are allowed.

➲ List interface in java has four concrete subclasses. They are ArrayList, LinkedList, Vector, and Stack. These four subclasses implements the list interface.

ArrayList and LinkedList are widely used in Java programs to create a list. The Vector class is deprecated since JDK 5.

Java Collections Notes

Features of List Interface in Java

There are the following features of list in java. They are as follows:

1. The list allows storing duplicate elements in Java. JVM differentiates duplicate elements by using 'index'. Index refers to the position of a certain object in an array. It always starts at zero.

For example, assume that there is a list with size 10. Suppose the first element is 'a' at zero index position and the second element is also 'a' which is at 9th position. Thus, there are two 'a' elements in the list at positions 0 and 9 respectively.

So, JVM will differentiate between both elements in the list based on their numeral position of the index. Therefore, the index is very useful and plays an important role to differentiate objects.

2. In the list, we can add an element at any position.

3. It maintains insertion order. i.e. List can preserve the insertion order by using the index.

4. It allows for storing many null elements.

5. Java list uses a resizable array for its implementation. Resizable means we can increase or decrease the size of the array.

6. Except for LinkedList, ArrayList, and Vector is an indexed-based structure.

7. It provides a special Iterator called a ListIterator that allows accessing the elements in the forward direction using hasNext() and next() methods.

In the reverse direction, it accesses elements using hasPrevious() and previous() methods. We can add, remove elements of the collection, and

can also replace the existing elements with the new element using ListIterator.

How to create a List in Java?

To create a list in java, we can use one of its two concrete subclasses: ArrayList, and LinkedList. We will use ArrayList to create a list and test methods provided by list interface in the program section.

```
List p = new ArrayList();

List q = new LinkedList();

List r = new Vector();

List s = new Stack();
```

How to create Generic List Object in Java

After the introduction of Generic in Java 1.5, we can restrict the type of object that can be stored in the list. The general syntax for creating a list of objects with a generic type parameter is as follows:

```
1. List<data type> list = new ArrayList<data type>(); // General sysntax.



For example:

   a. List<String> list = new ArrayList<String>(); // Creating a list of objects of String type using ArrayList.

   b. List<Integer> list = new LinkedList<Integer>(); Creating a list of objects of Integer type using LinkedList.
```

```
c. List<String> list1 = new LinkedList<String>();

d. List<obj> list = new ArrayList<obj>(); // obj is the type of object.



For example:

   List<Book> list=new ArrayList<Book>(); // Book is the type of object.



2. Starting from Java 1.7, we can use a diamond operator.

   a. List<String> str = new ArrayList<>();

   b. List<Integer> list = new LinkedList<>();
```

Java List Initialization

After creating a list, we need to initialize the list by adding elements to it. There are three methods to initialize the list. They are as follows:

- Using Arrays.asList
- Using Normal way
- Using Anonymous Inner class

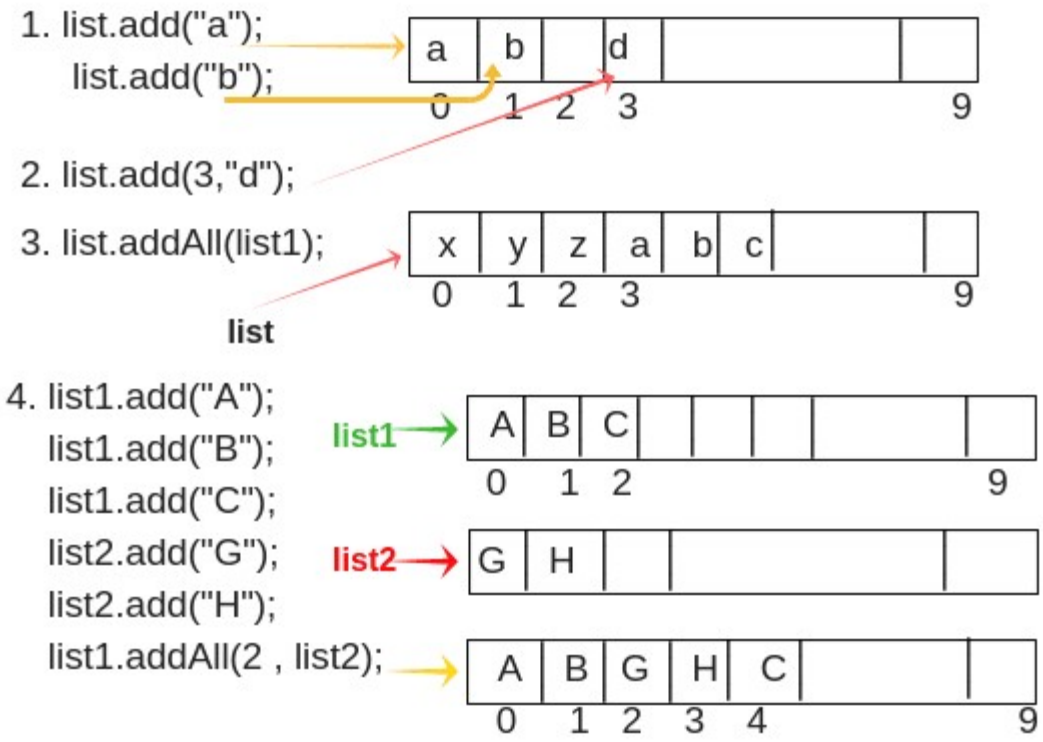
Go to this tutorial for more detail: ArrayList in Java

Java List Methods

Java List interface provides 10 specific and useful methods to the out of 15 methods specified by the collection interface that it extends. These methods can be used to initialize a list in java. They are as follows:

Java Collections Notes

1. boolean add(Object o): It starts to add the specified element from zero location. If the element is already present at zero location, it will add the next element in one position.



For example, consider 1 in the above figure.

```
list.add("a"); // Here, list is object reference variable.

list.add("b");
```

The return type of add() method is boolean and input type is an object.

2. void add(int index, Object o): This method adds/inserts the specified element at a particular position in the list. For example, suppose we want to add element “d” at 3rd position, we will call add(int index, Object o) method like this:

```
list.add(3,"d"); // It will add element "d" at 3rd position as shown in figure.
```

3. boolean addAll(Collection c): Here, Collection c represents a group of elements. This method is used to add/insert a group of elements at the end of the last element.

For example, suppose we want to add three elements x, y, and z at positions 0, 1, and 2 respectively in a list.

```
list.add("x");

list.add("y");

list.add("z");
```

Now we will create a list of another group of elements like this:

```
list1.add("a");

list1.add("b");

list1.add("c");

list.addAll(list1); // It will add group of elements at the end of the last element in the list. The last element is z. So, after z, it will add list1 as shown in above figure.
```

4. boolean addAll(int index, Collection c): This method is used to add/insert a group of elements at a particular position in the list and shift the subsequent elements to the right by increasing their indices. For example, suppose we want to add three elements at positions 1, 2, and 3 respectively in a list using list1 reference variable.

```
list1.add("A");

list1.add("B");

list1.add("C");
```

Now we will create a list of another group of elements using list2 reference variable.

```
list2.add("G");

list2.add("H");
```

Assume that we want to add this group of elements at position 2 of the list1. So, we will call addAll(int index, Collection c) method like this:

```
list1.addAll(2, list2); // You will see that element C is shifted to right at position 4 as shown in the figure.
```

5. object remove(int index): It is used to remove an element at a specified position in the list. For example, consider the above figure.

```
list1.remove(2); // It will remove the element at 2nd position and element D which is at 3rd position, will be shifted to the left at the 2nd position. The output will be A, B, D.
```

6. object get(int index): This method is used to return element/object stored at a specified position in the list. The return type of get() method is Object and input type is int[index of List].
For example:

```
list1.get(2); // Output will be 'C'.
```

7. int indexOf(Object o): It is used to return the index of a particular element of the first occurrence in the list. If the element is not present in the list then it will return -1. It takes as an argument as an element and returns as an integer value of that element as it is index value.
For example, suppose an element 'A' is present at position 0 and the same element is also present at position 9 in the list.

```
list.indexOf("A"); // It will return integer value of an element "A" of first occurrence. i.e from zero position, not from position 9. So the output is 0.
```

8. int lastIndexOf(Object o): It returns the index of the last occurrence of a specified element in the list. If the list does not contain that particular element, it will return -1.
For example:

```
list.lastIndexOf("A"); // Output will be 9.
```

9. object set(int index, Object o): This method replaces the existing element at the specified position in the list with new specified element.
For example:

```
list1.set(2, "Z");
```

10. ListIterator listIterator(): It returns listIterator of the elements in the list in a proper sequence.

11. ListIterator listIterator(int): This method returns a listIterator of the elements in the list in proper sequence, starting at the specified position in the list.

List Example Programs

Let's take different kinds of example programs based on all the above methods to understand better.

Let's create an example program where we will add both integer and string elements together. So, we will not use generic in this program.

Program source code 1:

```
package listPrograms;

import java.util.ArrayList;

import java.util.List;

public class AddEx

{

public static void main(String[] args)

{

// Create a List.

   List al = new ArrayList(); // Here, there is no use of generic. So, no type safety. We can add both integer and string elements.
```

```
// Adding elements using add() method with reference variable al.
   al.add(10);
   al.add(20);
   al.add(30);
   al.add(40);
   al.add("Shubh");

// Adding element to 4th position.
   al.add(4, 35);

// Adding element to 5th position.
   al.add(5, 45);

  System.out.println("Elements after adding: " +al);
 }
}
Output:
       Elements after adding: [10, 20, 30, 40, 35, 45, Shubh]
```

Java Collections Notes

Let's create a program where we will create a list 1 and another list 2. We will add elements of list 2 into list 1 at position 2. Look at the source code to understand better.

Program source code 2:

```
package listPrograms;

import java.util.ArrayList;

import java.util.List;

public class AddAllEx

{

public static void main(String[] args)

{

// Create a list1 of only String type. This means that Compiler will give errors if we try to put any elements other than String type.

   List<String> al = new ArrayList<String>();

   al.add("Apple");

   al.add("Mango");

   al.add("Orange");

   al.add("Grapes");

   System.out.println("List1 contain: " +al);



// Create another List2 of String type.

   List<String> al2 = new ArrayList<String>();
```

```
    al2.add("11");

    al2.add("12");

    al2.add("13");

  System.out.println("List2 contain :-"+al2);



// Adding List2 in List1 at 2nd position(i.e index=2) using addAll() method.

   al.addAll(2, al2);

  System.out.println("List1 after adding List2 at 2nd position :-"+al);

  }

}
```

Output:

```
    List1 contain: [Apple, Mango, Orange, Grapes]

    List2 contain: [11, 12, 13]

    List1 after adding List2 at 2nd position: [Apple, Mango, 11, 12, 13, Orange, Grapes]
```

Program source code 3:

```
package listPrograms;

import java.util.ArrayList;

import java.util.List;

public class IndexOfEx

{
```

```
public static void main(String[] args)

{

 List al = new ArrayList();

  al.add("AA");

  al.add("BB");

  al.add("CC");

  al.add("DD");

  al.add("EE");

  al.add("FF");



// To find the Index of any particular element, use obj.indexOf(object o) method.

   System.out.println("Index of CC: "+al.indexOf("CC"));

   System.out.println("Index of FF: "+al.indexOf("FF"));

 }

}

Output:

      Index of CC: 2

      Index of FF: 5
```

Program source code 4:

```
package listPrograms;
```

```
import java.util.ArrayList;

import java.util.List;

public class GetMethodEx

{

 public static void main(String[] args)

 {

   List al = new ArrayList();

// Adding Element using reference variable al.

   al.add("pen");

   al.add("pencil");

   al.add("ink");

   al.add("notebook");

   al.add("book");

   al.add("paper");



// Now call get(int index) method to get elements from specified index and print them.

   System.out.println("First Element: " +al.get(0));

   System.out.println("Fourth Element: " +al.get(3));

  }

}
```

```
Output:

    First Element: pen

    Fourth Element: notebook
```

Java List vs ArrayList

List is an interface whereas ArrayList is an implementation class that implements the List interface.

When to use List?

There are the following points for using list in java application that should be to keep in mind. They are:

1. List can be used when we want to allow or store duplicate elements.
2. It can be used when we want to store null elements.
3. When we want to preserve my insertion order, we should go for list.

ArrayList in Java | ArrayList Methods, Example

ArrayList in Java is a resizable array that can grow or shrink in the memory whenever needed. It is dynamically created with an initial capacity.
It means that if the initial capacity of the array is exceeded, a new array with larger capacity is created automatically and all the elements from the current array are copied to the new array.

Elements in ArrayList are placed according to the zero-based index. That is the first element will be placed at 0 index and the last element at index (n-1) where n is the size of ArrayList.

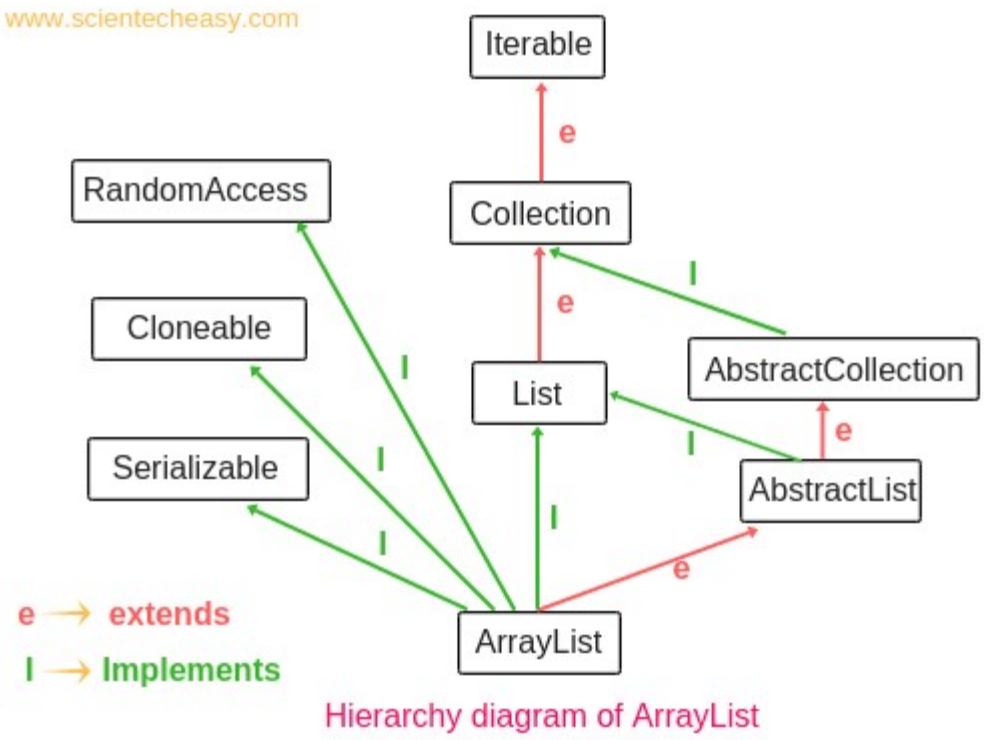
Java Collections Notes

Java ArrayList uses a dynamic array internally for storing the group of elements or data.

The capacity of ArrayList does not shrink automatically. When elements are removed from the list, the size of array list can be shrunk automatically but not capacity.

Hierarchy Diagram of ArrayList Class in Java

The hierarchy diagram of ArrayList class is shown in the below figure.



Hierarchy diagram of ArrayList

As shown in the above hierarchy diagram, ArrayList class implements the List interface and extends AbstractList (Abstract class) which implements List interface.
Java ArrayList class also implements 3 marker interfaces: Random Access, Cloneable, and Serializable.

Java Collections Notes

A marker interface is an interface that does not have any methods or any member variables. It is also called an empty interface because of no field or methods.

Random Access Interface

1. RandomAccess Interface is a marker interface that does not define any method or member. It is introduced in Java 1.4 version for optimizing the list performance.

2. RandomAccess interface is present in java.util package.

3. ArrayList class implements a random access interface so that we can access any random element at the same speed. For example, suppose there is a group of one crore objects in the array list. Assume that the first element is x, 10th element is y, and 1st crore element is z.

Now assume that first element x can be accessed within only 1 sec. Due to the implementation of random access interface, the 10th element and 1st crore element can also be accessed within 1 sec.

Thus, Any random element we can access with the same or constant speed. Therefore, if our frequent operation is retrieval operation then ArrayList is the best choice.

Serializable Interface

1. A serializable interface is a marker interface that is used to send the group of objects over the network. It is present in the java.io package.

2. It helps in sending the data from one class to another class. Usually, we use collections to hold and transfer objects from one place to another place.

To provide support for this requirement, every collections class already implements Serializable and Cloneable.

Cloneable Interface

1. A cloneable interface is present in java.lang package.

2. It is used to create exactly duplicate objects. When the data or group of objects came from the network, the receiver will create duplicate objects.

The process of creating exactly duplicate objects is known as cloning. It is a very common requirement for collection classes.

Features of ArrayList in Java

There are several features of ArrayList class in Java. They are as follows:

1. Resizable-array: ArrayList is a resizable array or growable array that means the size of ArrayList can increase or decrease in size at runtime. Once ArrayList is created, we can add any number of elements.
2. Index-based structure: It uses an index-based structure in java.
3. Duplicate elements: Duplicate elements are allowed in the array list.
4. Null elements: Any number of null elements can be added to ArrayList.
5. Insertion order: It maintains the insertion order in Java. That is insertion order is preserved.
6. Heterogeneous objects: Heterogeneous objects are allowed everywhere except TreeSet and TreeMap. Heterogeneous means different elements.
7. Synchronized: ArrayList is not synchronized. That means multiple threads can use the same ArrayList objects simultaneously.
8. Random Access: ArrayList implements random access because it uses an index-based structure. Therefore, we can get, set, insert, and remove elements of the array list from any arbitrary position.
9. Performance: In ArrayList, manipulation is slow because if any element is removed from ArrayList, a lot of shifting takes place.
For example, if an array list has 500 elements and we remove 50th elements then the 51st element will try to acquire that 50th position, and likewise all elements. Thus, it consumes a lot of time-shifting.

Java ArrayList Constructor

Java ArrayList class provides three constructors for creating an object of ArrayList. They are as:

- ArrayList()
- ArrayList(int initialCapacity)
- ArrayList(Collection c)

Creating an object of ArrayList class in Java is very simple. First, we will declare an array list variable and call the array list constructor to instantiate an object of ArrayList class and then assign it to the variable.

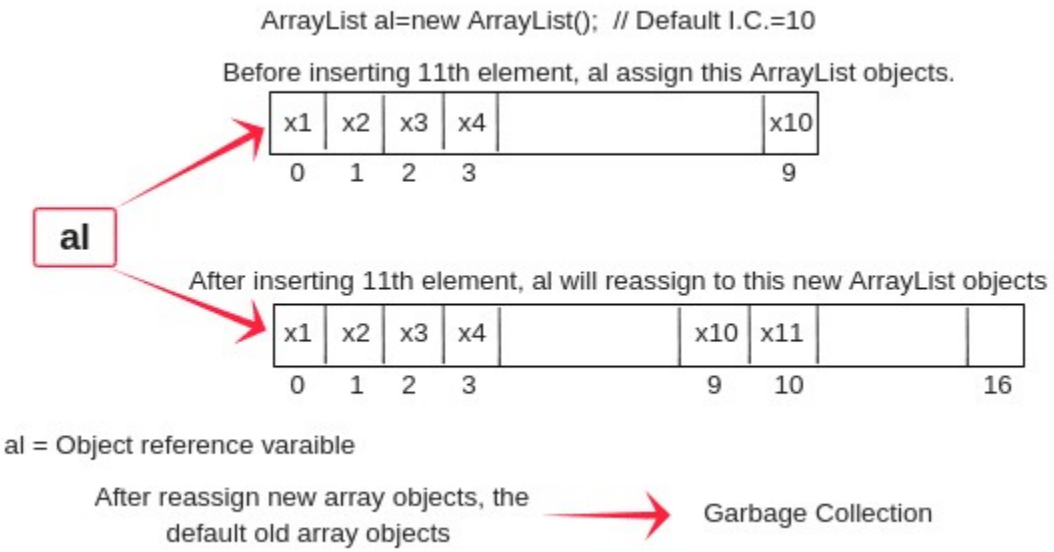
We can create an object of ArrayList class in java by using any one of three constructors. Let's see one by one.

1. The syntax for creating an instance of ArrayList class is as:

```
ArrayList al = new ArrayList();
```

It creates an empty ArrayList with a default initial capacity of 10. In this array list, we can store only 10 elements as shown in the below diagram.

Java Collections Notes



Suppose we insert the 11th element at 10th position into an array list, what will happen internally?

Once ArrayList is reached its maximum capacity, the ArrayList class automatically creates a new array with a larger capacity.

New capacity = (current capacity*3/2) + 1 = 10*3/2 + 1 = 16
After creating a new array list with a larger capacity, all the existing elements are copied into the new array list and then adds the new element into it. ArrayList class reassigns the reference to the new array objects as shown in the above figure.

The old default array list with the collection of objects is automatically gone into the garbage collection. Similarly, If we try to insert 17th element, new capacity= 16*3/2 + 1 = 25.

2. We can also initialize the capacity at the time of creating ArrayList object. The syntax for creating an instance of ArrayList class with initial capacity is as:

```
ArrayList al = new ArrayList(int initialCapacity);
```

It creates an empty ArrayList with initial capacity. If you know the initial capacity, you can directly use this method. Thus, we can improve the performance of the system by default.

For example, suppose our requirement is to add 500 elements in the array list, we will create ArrayList object like this:

```
ArrayList list2 = new ArrayList(500);
```

3. We can also create an object of ArrayList class by initializing a collection of elements into it. The syntax is as follow:

```
ArrayList al = new ArrayList(Collection c);
```

It creates an ArrayList object by initializing elements of collection c.

For example:

```
ArrayList list3 = new ArrayList(list1); // list1 is elements of collection.
```

Creating Generic ArrayList Object in Java

Java 1.5 version or later also provides us to specify the type of elements in the ArrayList object. For example, we can create a generic String ArrayList object like this:

```
   ArrayList<String> al = new ArrayList<String>(); // It can be used to store only String type.



// The advantage of specifying a type is that when we try to add another type of element, it will give compile-time error.



or,
```

```
Creating a Generic ArrayList object can also be done in separate lines like this:

  ArrayList<String> arlist;

       arlist = new ArrayList();
```

Note: We cannot use primitive data types as a type. For example, ArrayList<int> is illegal.

Java ArrayList Initialization

There are three methods to initialize the array list in Java. They are as follows:

1. Using Arrays.asList: The syntax to initialize an ArrayList using asList() method is as follows:

```
ArrayList<Type> list = new ArrayList<Type>(Arrays.asList(Object o1, Object o2, .. so on));



For example:

 ArrayList<String> ar = new ArrayList<String>(Arrays.asList("A", "B", "C"))
```

2: Using normal way: This is a popular way to initialize ArrayList in java program. The syntax to initialize array list is as:

```
ArrayList<Type> obj = new ArrayList<Type>();

 obj.add("Obj o1");

 obj.add("Obj o2");

 and so on.
```

3. Using Anonymous Inner class: The syntax for initialization of ArrayList in java using anonymous inner class is as:

```
ArrayList<Type> arl = new Arraylist<Type>() {{

  add(Object o1);

  add(Object o2);

  add(Object o3);

  . . . . . .

  . . . . . .

 }};
```

ArrayList Methods in Java

ArrayList also provides some useful methods. They are as follows:

1. boolean add(Object o): This method is used to add an element at the end of array list. For example, if you want to add an element at the end of the list, you simply call the add() method like this:

```
list.add("Shubh"); // This will add "Shubh" at the end of the list.
```

2. boolean addAll(Collection c): This method is used to add a group of elements in a particular collection at the end of the list. For example, suppose we have a group of elements in the list2 and want to add at the end of the list1, we will call this method like this:

```
 list1.addAll(list2);
```

3. boolean addAll(int index, Collection c): This method is used to add a group of elements at a specified position in a list. For example:

```
list1.addAll(2, list2): // Adding all elements of list2 at position index 2 in list1
```

4. void add(int index, Object o): It is used to add an element at a particular position index in the list. For example:

```
list.add(3, "a"); // Adding element 'a' at position index 3 in list.
```

5. void addAll(int index, Object o): It is used to add a specific element at a particular position in the list. For example, suppose we want to a specific element "Shubh" at a position 2 in the list, we will call add(int index, Object o) method like this:

```
list.add(2,"Shubh"); // This will add "Shubh" at the second position.
```

Let's take an example program based on the above ArrayList add() methods.

Program source code 1:

```
package ArrayListTest;

import java.util.ArrayList;

public class AddExample

{

public static void main(String[] args)

{

// Create an object of the non-generic ArrayList.

   ArrayList al = new ArrayList(); // list 1 with default capacity 10.

    al.add("A");

    al.add("B");

    al.add(20);

    al.add("A");
```

```
    al.add(null);

   System.out.println(al);



// Create an object of another non-generic ArrayList.

   ArrayList al1 = new ArrayList(); // List 2.

    al1.add("a");

    al1.add("b");

    al1.add("c");



// Call addAll(Collection c) method using reference variable al to add all elements at the
end of the list1.

    al.addAll(al1);

    System.out.println(al);



// Call addAll(int index, Collection c) method using reference variable al1 to add all
elements at specified position 2.

    al1.addAll(2, al);

   System.out.println(al1);

  }

}
```

Output:

```
[A, B, 20, A, null]

[A, B, 20, A, null, a, b, c]

[a, b, A, B, 20, A, null, a, b, c, c]
```

In the preceding example program, we have used duplicate elements, heterogeneous elements (i.e String and integer), and a null element. You will also notice that when a specified element is added at a particular position, the right side element is shifted one position right.

After shifting, the object reference variable will reassign a new array list as shown in the above picture. Due to shifting, ArrayList is also time-consuming.

In the output, we are getting square bracket notation for this ArrayList. This is because when we try to print an object reference variable, internally it calls toString() method like this:

```
System.out.println(al); ------> System.out.println(al.toString());
```

5. boolean remove(Object o): It removes the first occurrence of the specified element from this list if it is present.
6. void remove(int index): This method removes the element from a particular position in the list. Look at the examples below.

```
list.remove("A");

list.remove(2); //  It will remove element from position 2.
```

7. void clear(): The clear() method is used to remove all elements from an array list.
8. void set(int index, Object o): The set() method replaces element at a particular position in the list with the specified element. For example, suppose we want to replace an element “A” at a position 2 with an element “a” in the list, we will have to call this method as:

```
list.set(2, "a");
```

Let's take an example program where we will remove an element using remove() method from the list.

Program source code 2:

```
package ArrayListTest;

import java.util.ArrayList;

public class RemoveEx

{

public static void main(String[] args)

{

// Create a generic Arraylist object of String type.

// This means the compiler will show an error if we try to put any other element than String.

    ArrayList<String> al = new ArrayList<String>(); // Default capacity is 10.



// Adding elements of String type.

     al.add("A");

     al.add("B");

     al.add("C");

     al.add("D");

     al.add(null);

     al.add("D");
```

```
  System.out.println(al);



// Call remove() method to remove element D.

    al.remove("D"); // removes the first occurrence of the specified element D at position 3, not from the position 5.

    System.out.println(al);

    al.remove(3);

    System.out.println(al);



// Call set method to replace the element D with a null element at position 3.

    al.set(3, null);

    System.out.println(al);

  }

}
```

Output:

```
      [A, B, C, D, null, D]

      [A, B, C, null, D]

      [A, B, C, D]

      [A, B, C, null]
```

In this example program, you will observe that when an element is removed or deleted from an array, the deleted element becomes null but the empty slot occupied by the deleted element stays in the array.

Any subsequent elements at the right side in the array are automatically shifted towards one position left to fill empty slot that was occupied by deleted element and object reference variable 'al' will be reassigned to the new array list like as shown in the above diagram.

9. Object get(int index): It returns the element at the specified position in this list.

10. int size(): It returns the number of elements of the list. Size means the number of elements present in the array list. Capacity means the capability to store elements.

11. boolean contains(Object o): It returns true if a specified element is present in the list, else, returns false. Look at the examples below.

```
String str = list.get(2);

int numberOfElements = list.size();

list.contains("A");
```

Let's see an example program based on these methods.

Program source code 3:

```
package arrayListPrograms;

import java.util.ArrayList;

public class ArrayListTest

{

public static void main(String[] args)

{

 ArrayList al = new ArrayList();
```

```
  al.add("Apple");

  al.add("Orange");

  al.add("Banana");

  al.add("Gauva");

System.out.println(al);



// Call get() method using object reference variable 'al' to get the specified element.

// Since return type of get() method is String. Therefore, we will store it by using a
fruitsName variable with data type String.

     String fruitsName = al.get(2);

     System.out.println(fruitsName);



// Call size() method to get the number of elements present in the list.

// Since return type of size method is an integer. Therefore, we will store it by using
variable numberOfElements with data type integer.

     int numberOfElements = al.size();

     System.out.println(numberOfElements);



// Check apple element is present or not.

   boolean check = al.contains("Apple");

   System.out.println(check);
```

```
 }

}

Output:

      [Apple, Orange, Banana, Gauva]

      Banana

      4

      true
```

12. int indexOf(Object o): It is used to get the index of the first occurrence of the specified element, if the element is not present in the list, it returns the value -1.

13. int lastIndexOf(Object o): It is used to get the index of the last occurrence of the specified element in the list. If the element is not present in the list, it returns -1. For example:

```
int pos = list.indexOf("Apple");

int lastPos = list.lastIndexOf(20);
```

Program source code 4:

```
package ArrayListTest;

import java.util.ArrayList;

public class Test

{

public static void main(String[] args)

{

 ArrayList<Integer> list = new ArrayList<Integer>();
```

```
 list.add(10);

 list.add(20);

 list.add(30);

 list.add(40);



System.out.println(list);

int pos = list.indexOf(30);

System.out.println(pos);



int lastPos = list.lastIndexOf(40);

System.out.println(lastPos);

}

}
```

Output:

```
[10, 20, 30, 40]

23
```

How do we manually increase or decrease current capacity of ArrayList?

Java Collections Notes

1. ensureCapacity(): This method is used to increase the current capacity of ArrayList. Since the capacity of an array list is automatically increased when we add more elements. But to increase manually, ensureCapacity() method of ArrayList class is used.

2. trimTosize(): The trimTosize() method is used to trim the capacity of ArrayList to the current size of ArrayList.

```
ArrayList<String> list = new ArrayList<String>(); // Here, list can hold 10 elements.(Default initial capacity).

 list.ensureCapacity(20); // Now it can hold 20 elements.



 list.trimTosize();
```

When to use ArrayList in Java?

ArrayList can be used in an application program when

- We want to store duplicate elements.
- We want to store null elements.
- It is more preferred in Java when getting of the element is more as compared to adding and removing elements.
- We are not working in the multi-threading environment in Java because ArrayList is non-synchronized.

Enumeration in Java | Iterators in Java

Enumeration in Java | In the previous tutorial, we knew that ArrayList in Java is a resizable array that stores elements in the memory dynamically. If the capacity of the array is exceeded then a new array with a larger capacity will be created and all the existing elements from the current array will be copied into the new array.

Now in this tutorial, we will learn what is iterators in java, and their types.

Java Collections Notes

Iterators in Java

Iterators in Java are used to retrieve the elements one by one from a collection object. They are also called cursors in java.
Let's understand this concept with a realtime example.

Suppose that there are 20 apples in a box. If I ask you how will you eat all these 20 apples? one by one, or all apples at once time.

I hope that your answer will be definitely one by one. That is, you will eat the first apple. After that, you will eat the second apple, and so on.

Similarly, you assume that box is a collection, and apples in the box are elements of the collection. So if we want to get these elements one by one, we will require some iterators or cursors to retrieve elements one by one from the collection object.

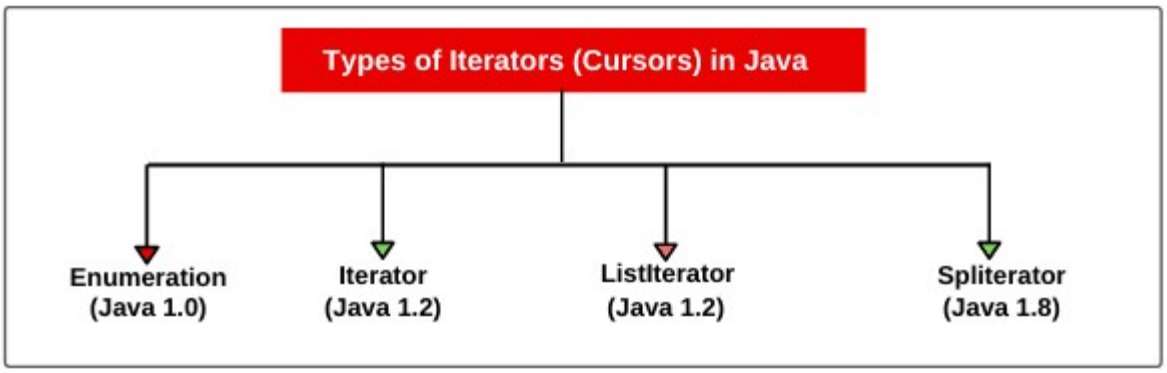
Therefore, we need to learn iterators or cursors concepts in Java.

Types of Iterators in Java

There are four types of iterators or cursors available in Java. They are as follows:

- Enumeration
- Iterator
- ListIterator
- Spilterator

Java Collections Notes



Each Java iterator has some advantages and limitations. In this tutorial, we will discuss some basic points about Enumeration. We will discuss Iterator, ListIterator, and Spliterator in the coming tutorials one by one.

Enumeration in Java

Enumeration is the first iterator that was introduced in Java 1.0 version. It is located in java.util package. It is a legacy interface that is implemented to get elements one by one from the legacy collection classes such as Vector and Properties.

Legacy classes are those classes that are coming from the first version of Java. Early versions of Java do not include collections framework. Instead, it defined several classes and one interface for storing objects.
When collections came in the Java 1.2 version, several of the original classes were re-engineered to support the collection interfaces.

Thus, they are fully compatible with the framework. These old classes are known as **legacy classes**. The legacy classes defined by java.util are Vector, Hashtable, Properties, Stack, and Dictionary. There is one legacy interface called Enumeration.
Enumeration is read-only. You can just read data from the vector. You cannot remove it from the vector using Enumeration.

How to create Enumeration object in Java?

Since enumeration is an interface so we cannot create an object of enumeration directly. We can create an object of enumeration by calling elements() method of the Vector class.

The syntax for creating an object of enumeration in java is as follows:

Syntax:

```
public Enumeration elements() // Return type is Enumeration.



For example:

   Enumeration e = v.elements(); // Here, v is a vector class object.
```

Now you will think that how can create object of Enumeration Interface?

Actually, we cannot create the object of Interface directly or indirectly. When we create an object of interface like this:

```
Enumeration e = v.elements();
```

Internally, elements() method will be executed in the vector class. See below the snippet code.

```
class Vector

{

 elements()

 {

  class  implements Enumeration

  {
```

```
    . . . . . .

    . . . . . .

  }

    return (implemented class object);

 }

}
```

You can see the above code that within the elements() method, there is a class that implements enumeration and the implemented class object is gone to return.

We can print corresponding internally implemented class name on the console by below snippet code.

```
Enumeration e = v.elements();

System.out.println(e.getClass().getName()); // Output: vector$1
```

vector$1 ➡ Inside vector, $ is the inner class name and 1 is the convention which is applicable for anonymous class.

Thus, when we are creating an enumeration object, internally, we are creating an enumeration implemented class object.

Similarly, the same thing happens in the case of Iterator and ListIterator. Let's see the code structure.

Program source code 1:

```
import java.util.Enumeration;

import java.util.Iterator;

import java.util.ListIterator;
```

```
import java.util.Vector;



public class InnerClassName

{

public static void main(String[] args)

{

 Vector v = new Vector();

 Enumeration e = v.elements();

 Iterator itr = v.iterator();

 ListIterator litr = v.listIterator();



System.out.println(e.getClass().getName());

System.out.println(itr.getClass().getName());

System.out.println(litr.getClass().getName());

 }

}

Output:

        java.util.Vector$1 java.util.Vector$Itr java.util.Vector$ListItr
```

Vector$Itr ➡ Itr is the inner class present inside the vector which implements the Iterator interface.

Vector$ListItr ➡ ListItr is the inner class inside the vector which implements the ListIterator interface.

Methods of Enumeration in Java

The Enumeration interface defines the following two methods. They are as follows:

1. public boolean hasMoreElements(): When this method is implemented, hasMoreElements() will return true If there are still more elements to extract and false if all the elements have been enumerated.
2. public Object nextElement(): The nextElement() method returns next element in the enumeration. It will throw NoSuchElementException when the enumeration is complete.
Let's take some example programs based on concepts of these methods to understand better.

Program source code 2:

```
import java.util.Enumeration;

import java.util.Vector;

public class EnumerationTest

{

public static void main(String[] args)

{

// Create object of vector class without using generic.

   Vector v = new Vector();
```

```
// Add ten elements of integer type using addElement() method. For this we will use for loop.

   for(int i = 0; i < = 10; i++)

   {

      v.addElement(i);

   }

   System.out.println(v); // It will print all elements at a time like this [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



// Now we want to get elements one by one. So, we will require Enumeration concept.



// Create object of Enumeration by calling elements() method of vector class using object reference variable v.



// At the beginning, e (cursor) will point to index just before the first element in v.

   Enumeration e = v.elements();



// Checking the next element availability using reference variable e and while loop.

   while(e.hasMoreElements())

   {

// Moving cursor to next element.
```

```
    Object o = e.nextElement();

    Integer i = (Integer)o; // Here, Type casting is required because the return type of
nextElement() method is an object. Therefore, it's compulsory to require type casting.



   System.out.println(i);

  }

 Enumeration en = v.elements();

 while(en.hasMoreElements())

 {

  Object o = en.nextElement();

  Integer it = (Integer)o;

// Getting even elements one by one.

  if(it % 2 == 0)

  {

   System.out.println(it);

  }

 }

}

}

Output:
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

0 1 2 3 4 5 6 7 8 9 10 0 2 4 6 8 10
```

Now let's perform the same program using the generic concept.

Program source code 3:

```
import java.util.Enumeration;

import java.util.Vector;

public class EnumerationTest

{

public static void main(String[] args)

{

// Create an object of vector class using generic.

 Vector<Integer> v = new Vector<Integer>();

 for(int i=0; i<=10; i++)

 {

   v.addElement(i);

 }

System.out.println(v);

Enumeration e = v.elements();

while(e.hasMoreElements())

{
```

```
  Integer i = (Integer)e.nextElement(); // Direct type casting in one step.

  System.out.println(i);

 }

Enumeration en = v.elements();

while(en.hasMoreElements())

{

  Integer it = (Integer)en.nextElement();

  if(it % 2 == 0)

  {

    System.out.println(it);

  }

 }

 }

}

Output:

      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

      0 1 2 3 4 5 6 7 8 9 10 0 2 4 6 8 10
```

Limitation of Enumeration

Java Collections Notes

There are many limitations of using enumeration interface in java. They are as follows:

1. Enumeration concept is applicable for only legacy class. Hence, it is not a universal cursor.
2. We can get only read operation by using the enumeration. We cannot perform the remove operation.
3. We can iterate using enumeration only in the forward direction.
4. Java is not recommended to use enumeration in new projects.

To overcome these limitations, We should go for the next level Iterator concept in Java.

Iterator in Java | Methods, Iterable Interface, Example

An **iterator in Java** is a special type of object that provides sequential (one by one) access to the elements of a collection object.
In simple words, it is an object that allows us to get and process one element at a time. It is introduced in Java 1.2 Collections Framework.

An Iterator object implements Iterator interface which is present in java.util.Iterator package. Therefore, to use an Iterator, you must import either java.util.Iterator or java.util.*.

Iterator in Java is used in the Collections Framework to retrieve elements sequentially (one by one). It is called **universal Iterator** or cursors.
It can be applied to any collection object. By using Iterator, we can perform both read and remove operations.

Iterable Interface in Java

The Collection interface extends Iterable interface that is present at the top of the collection hierarchy. The iterable interface is present in

java.lang.Iterable package. It provides a uniform way to retrieve the elements one by one from a collection object.
It provides just one method named iterator() which returns an instance of Iterator and provides access one by one to the elements in the collection object.

The general syntax of iterator() method is as follows:

```
iterator() : Iterator // Return type is Iterator.
```

How to create Iterator object in Java?

Iterator object can be created by calling iterator() method which is present in the iterable interface. The general syntax for creating Iterator object is as follows:

```
a) Iterator itr = c.iterator(); // c is any collection object.

b) Iterator<Type> itr = c.iterator(); // Generic type.



For example:

    Iterator<String> itr = c.iterator();

    Iterator<Integer> itr = c.iterator();
```

How Java Iterator works internally?

The basic working mechanism for using Java Iterator is shown in the flow diagram.

Java Collections Notes

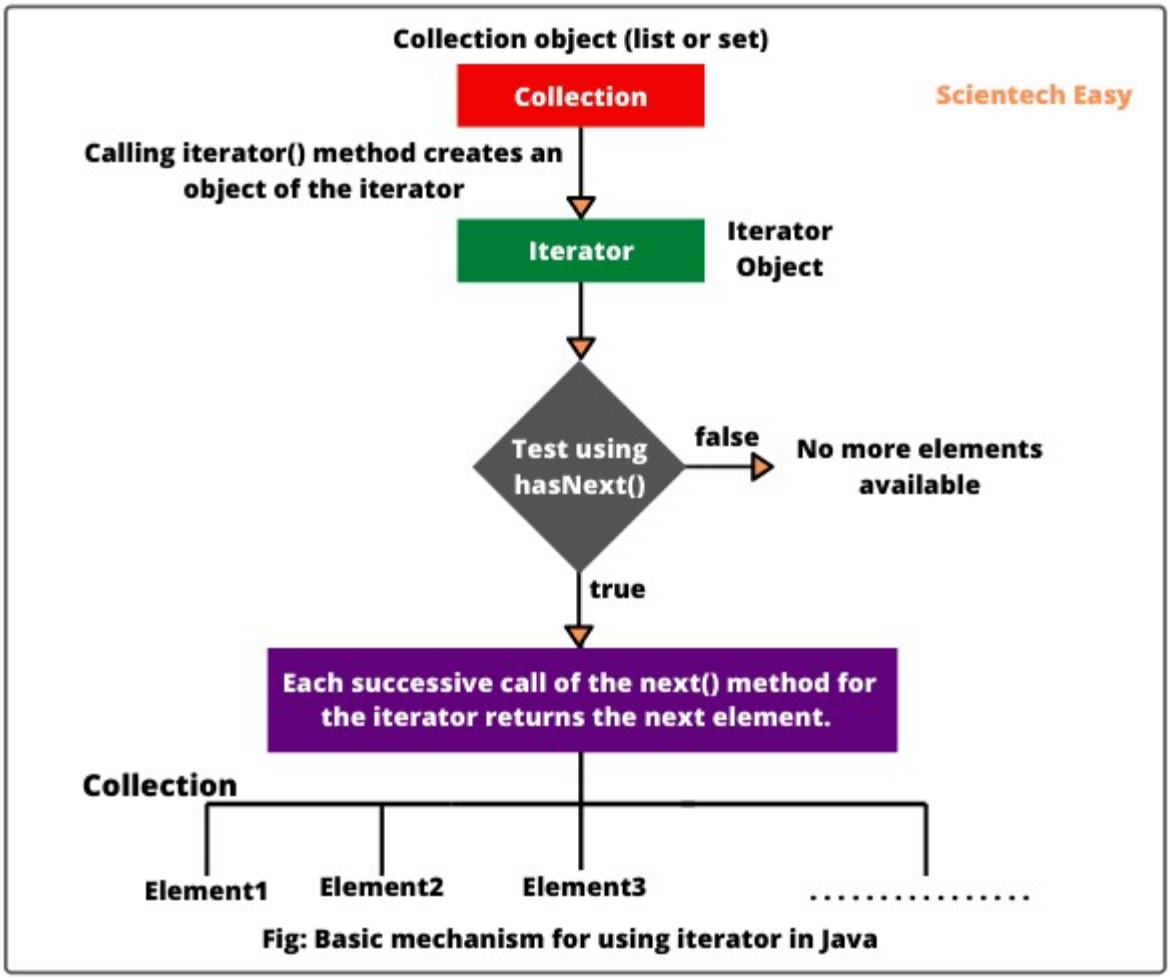


Fig: Basic mechanism for using iterator in Java

Let's take an example program to understand how Java Iterator works internally?

Look at the source code of a simple program.

Program source code 1:

```
import java.util.ArrayList;

import java.util.Iterator;

public class IteratorTest

{

public static void main(String[] args)

{
```

```
 ArrayList<String> al = new ArrayList<String>();

// Adding elements in the array list.

   al.add("A");

   al.add("B");

   al.add("C");

   al.add("D");

   al.add("E");

   al.add("F");



Iterator<String> itr = al.iterator();

while (itr.hasNext())

{

 String str = itr.next();

 System.out.print(str + " ");

 }

}

}
```

Output:

A B C D E F

Java Collections Notes

When we create an iterator for the collection elements, the iterator looks like the following.

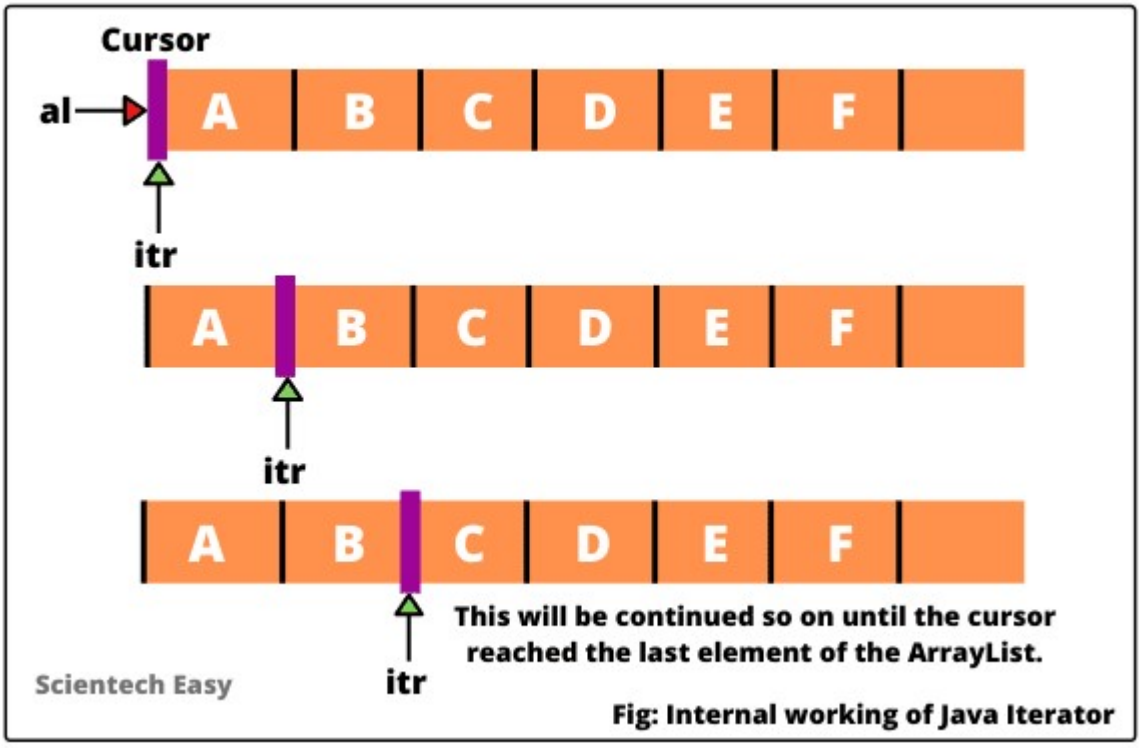


Fig: Internal working of Java Iterator

1. In the beginning, an Iterator points at right before the first element of the collection as shown in the above figure.
2. When hasNext() method is called on this Iterator, it returns true because elements are present in the forward direction.

With the call to the next() method, it returns an element from the collection and sets the Iterator object to the next element on the next call of this method.

3. Again hasNext() method will be called to check that elements are present for iteration. Since elements are present for iteration in the forward direction, therefore, it will return true.

On call of next() method, it will return the next element from the collection and will set the Iterator object to the next element on the next call of this method.

4. Calling of hasNext() and next() method will be continued until the pointer moves to the last element.

5. When the pointer reached the last element of the ArrayList then the call to the hasNext() method will return false and the call to next() method will give an exception named NullPointerException.

Iterator Methods in Java

The Iterator interface provides three methods in Java. They are as follow:

1. public boolean hasNext(): This method will return true if the iteration has more elements to traverse (iterate) in the forward direction. It will give false if all the elements have been iterated.
2. public Object next(): The next() method return next element in the collection. It will throw NoSuchElementException when the iteration is complete.
3. public void remove(): The remove() method removes the last or the most recent element returned by the iterator. It must be called after calling the next() method otherwise it will throw IllegalStateException. Let's take some example programs based on these methods in an easy way and step by step. We will use the Iterator concept to traverse all elements in the ArrayList.

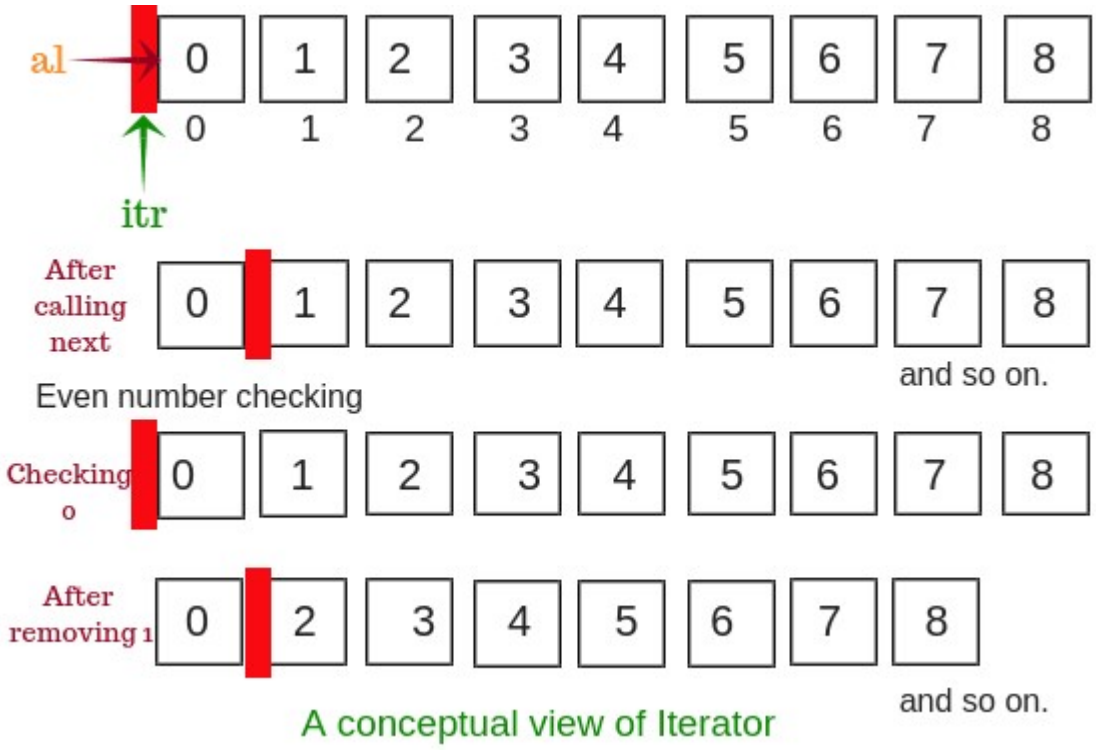
Program source code 2:

```
package iteratorsTest;

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorTest

{
```

```
public static void main(String[] args)

{

// Create an object of ArrayList of type Integer.

  ArrayList<Integer> al = new ArrayList<Integer>();

  for(int i = 0; i < = 8; i++)

  {

     al.add(i);

  }

System.out.println(al); // It will print all elements at a time.



// Create the object of Iterator by calling iterator() method using reference variable al.

// At the beginning, itr (cursor) will point to index just before the first element in al.

   Iterator<Integer> itr = al.iterator();



// Checking the next element availability using reference variable itr.

   while(itr.hasNext())

   {

// Moving cursor to next element using reference variable itr.

    Integer i = itr.next(); // Here, Type casting does not require due to using of generic with
Iterator.
```

```
   System.out.println(i);



// Removing odd elements.

   if(i % 2! = 0)

     itr.remove();

   }

  System.out.println(al);

 }

}
```

Look at the below picture to understand how elements are iterating in the above program.



```
Output:

    [0, 1, 2, 3, 4, 5, 6, 7, 8]

    0 1 2 3 4 5 6 7 8

    [0, 2, 4, 6, 8]
```

Let's create one more program for the practice where we will iterate all the elements in the ArrayList using iterator() method.

Program source code 3:

```
import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

public class IteratorTest {

public static void main(String[] args)

{

Collection<String> collection = new ArrayList<>();



// Adding elements in the array list.

   collection.add("Red");

   collection.add("Green");

   collection.add("Black");

   collection.add("White");

   collection.add("Pink");
```

```
Iterator<String> iterator = collection.iterator();



while (iterator.hasNext())

{

  System.out.print(iterator.next().toUpperCase() + " ");

}

System.out.println();

}

}

Output:

     RED GREEN BLACK WHITE PINK
```

Difference between Enumeration and Iterator

Both are useful to retrieve elements from a collection. But the main difference between Enumeration and iterator is with respect to functionality.

By using an enumeration, we can perform only read access but using an iterator, we can perform both read and remove operation.

Advantage of Iterator in Java

Java Collections Notes

Java Iterator has the following advantages. They are as follows:

- An iterator can be used with any collection classes.
- We can perform both read and remove operations.
- It acts as a universal cursor for collection API.

Limitation of Iterator in Java

Iterator has the following limitations or drawbacks. They are as:

- By using Enumeration and Iterator, we can move only towards forwarding direction. We cannot move in the backward direction. Hence, these are called single-direction cursors.
- We can perform either read operation or remove operation.
- We cannot perform the replacement of new objects.
- For example, suppose there are five mangoes in a box. Out of five, two mangoes are not good but we cannot replace those damaged mangos with new mangos.

To overcome the above drawbacks, we should use the ListIterator concept.

ListIterator in Java | Methods, Example

ListIterator in Java is the most powerful iterator or cursor that was introduced in Java 1.2 version. It is a bi-directional cursor.
Java ListIterator is an interface (an extension of Iterator interface) that is used to retrieve the elements from a collection object in both forward and reverse directions.

In simple words, it is an object by which we can iterate the elements of list in both forward and backward directions. It adds an additional six methods that reflect the bidirectional nature of a list iterator.

Java Collections Notes

By using ListIterator, we can perform different kinds of operations such as read, remove, replacement (current object), and the addition of the new elements.

Java ListIterator can be used for all List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc.

How to create ListIterator object in Java?

We can create a ListIterator object by calling listIterator() method of the List interface. The general syntax for creating ListIterator object in Java is as follows:

```
public ListIterator listIterator()  // return type is ListIterator.



Syntax for creating ListIterator object:

   ListIterator<Type> litr = l.listIterator(); // l is any list object and Type is type of objects being iterated.



For example:

  ListIterator<String> litr = l.listIterator(); // Get a full list iterator.



// Creating a list iterator object which will start to iterate at index 3 in the forward direction.

    ListIterator<String> litr = l.listIterator(3);
```

Key point:

Java Collections Notes

When we use ListIterator<String>, the return type of next() method is String. In general, the return type of next() method matches the ListIterator's type parameter (which indicates the type of the elements in the list).

Methods of ListIterator in Java

Since Java ListIterator interface extends the Iterator interface to add a bidirectional traversal of the list. Therefore, all the methods provided by Iterator are available by default to the ListIterator. ListIterator interface has a total of 9 methods. They are as follows:

Forward direction:

1. public boolean hasNext(): This method returns true if the ListIterator has more elements when iterating the list in the forward direction.
2. public Object next(): This method returns the next element in the list. The return type of next() method is Object.
3. public int nextIndex(): This method returns the index of the next element in the list. The return type of this method is an integer.

Backward direction:

4. public boolean hasPrevious(): It checks that list has more elements in the backward direction. If the list has more elements, it will return true. The return type is boolean.
5. public Object previous(): It returns the previous element in the list and moves the cursor position backward direction. The return type is Object.
6. public int previousIndex(): It returns the index of the previous element in the list. The return type is an Integer.

Other capability methods:

7. public void remove(): This method removes the last element returned by next() or previous() from the list. The return type is 'nothing'.

8. public void set(Object o): This method replaces the last element returned by next() or previous() with the new element.
9. public void add(Object o): This method is used to insert a new element in the list.
We will implement these methods one-by-one in different example programs in the below sections.

Let's take a simple example program based on these methods. For simplicity, we will iterate elements of the collection only in forward direction.

Program source code 1:

```
import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;



public class ListIteratorTest {

public static void main(String[] args)

{

 List<String> list = new ArrayList<>();

  list.add("A"); // Adding element A at index 0.

  list.add("B"); // Adding element B at index 1.

  list.add("C"); // Adding element C at index 2.



System.out.println("List: " + list);
```

```
// Create the list iterator object by calling listIterator() method.
// | in the comments indicates the position of iterator.
    ListIterator<String> iterator = list.listIterator(); // |ABC
    System.out.println("List Iterator in Forward Direction:");

// Call hasNext() method to check elements are present in forward direction.
    boolean elementsPresent = iterator.hasNext(); // Return true.
    System.out.println(elementsPresent);

 int indexA = iterator.nextIndex();
 String elementA = iterator.next(); // A|BC
 System.out.println("IndexA = " +indexA +" "+ "Element: " +elementA);

 int indexB = iterator.nextIndex();
 String elementB = iterator.next(); // AB|C
 System.out.println("IndexB = " +indexB +" "+ "Element: " +elementB);

 int indexC = iterator.nextIndex();
 String elementC = iterator.next(); // ABC|
```

```
 System.out.println("IndexC = " +indexC +" "+ "Element: " +elementC);



 boolean elementsPresent2 = iterator.hasNext(); // Return false because the iterator is at
the end of the collection.

 System.out.println(elementsPresent2);

 String element = iterator.next(); // It will throw NoSuchElementException because there
is not next element.

 }

}
```

Output:

```
List: [A, B, C]

List Iterator in Forward Direction:

true

IndexA = 0 Element: A

IndexB = 1 Element: B

IndexC = 2 Element: C

false

Exception in thread "main" java.util.NoSuchElementException
```

How Java ListIterator works internally?

Since ListIterator in Java works for iteration in both forward as well as backward directions, therefore, it is called a bi-directional Iterator.

For this, let's take an example program to understand the internal working of Java ListIterator. Look at the program source code.

Program source code 2:

```
import java.util.LinkedList;

import java.util.List;

import java.util.ListIterator;



public class ListIteratorTest {

public static void main(String[] args)

{

List<String> list = new LinkedList<>();

   list.add("A");

   list.add("B");

   list.add("C");



// Creating ListIterator object.

   ListIterator<String> listIterator = list.listIterator();



// Traversing elements in forwarding direction.
```

```
  System.out.println("Forward Direction Iteration:");



while(listIterator.hasNext())

{

  System.out.println(listIterator.next());

}

// Traversing elements in the backward direction. The ListIterator cursor is at just after
the last element.

   System.out.println("Backward Direction Iteration:");



while(listIterator.hasPrevious())

{

  System.out.println(listIterator.previous());

}

}}
```

Output:

```
      Forward Direction Iteration:

      A

      B

      C
```

Java Collections Notes

Backward Direction Iteration:

C

B

A

In the previous tutorial, we have already explained how Java Iterator works internally in the forwarding direction. However, ListIterator also works in the same way as Java Iterator.

If you do not know how Java Iterator works internally, go to this tutorial: Iterator in Java.
In this section, we will know how ListIterator works in a backward direction internally. Look at the below figure to understand better.

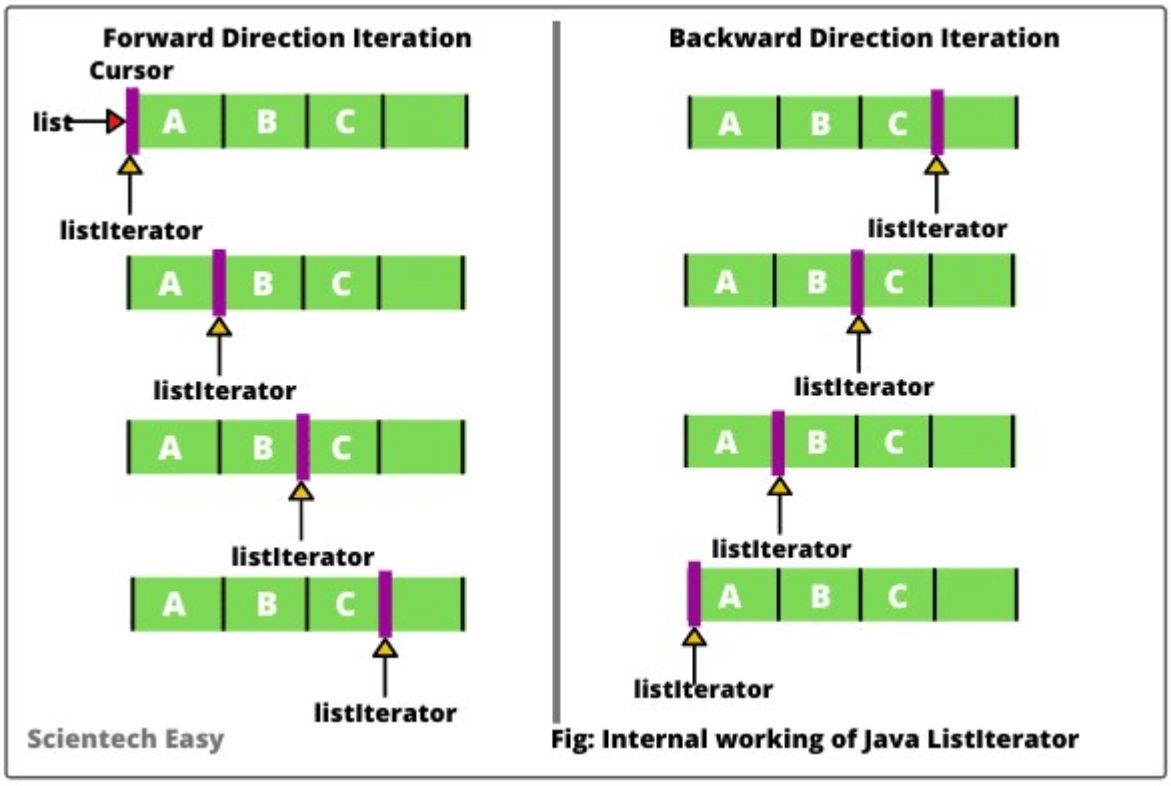


Fig: Internal working of Java ListIterator

1. Basically, ListIterator works in the backward direction the same as in the forward direction. When ListIterator's cursor reached right after to the last element in the list as shown in the above figure, the call to

hasPrevious() method checks that elements are present in the backward direction in the list. Since the elements are present in the backward direction in the list, so it will return true.

2. When previous() method is called, it returns the element and sets the position of the cursor for the next element in the backward direction. Look at the figure.

3. The call to hasPrevious() and previous() methods continue operations until ListIterator's cursor reaches the last element.

4. As soon as ListIterator's cursor points to the before the first element of the LinkedList, the hasPrevious() method returns a false value.

After observing the above figure, we can see that the hasPrevious() and previous() methods do the same task but in the opposite direction of the hasNext() and next() methods.

Thus, we can say that Java ListIterator iterates elements of the list in both forward as well as backward directions. Therefore, it is also known as **bi-directional cursor**.

Let's take another example program where we will get to the index of the next and previous element from the current position using its nextIndex() and previousIndex() methods. Look at the following source code.

Program source code 2:

```
import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;



public class ListIteratorTest {

public static void main(String[] args)

{
```

```
List<String> list = new ArrayList<>();

 list.add("Red");

 list.add("Green");

 list.add("Yellow");

 list.add("Orange");

 list.add("Blue");

 list.add("White");

System.out.println("List: " + list);



// Get the list iterator

 ListIterator<String> iterator = list.listIterator();



System.out.println();



System.out.println("List Iterator in Forward Direction:");

while (iterator.hasNext())

{

 int index = iterator.nextIndex();

 String element = iterator.next();

 System.out.println("Index = " + index + ", Element = " + element);
```

```
}
System.out.println();


System.out.println("List Iterator in Backward Direction:");


// Reuse the Java list iterator to iterate from the end to the beginning.
 while (iterator.hasPrevious())
 {
  int index = iterator.previousIndex();
  String element = iterator.previous();
  System.out.println("Index = " + index + ", Element = " + element);
 }
}
}
```

Output:

```
    List: [Red, Green, Yellow, Orange, Blue, White]


    List Iterator in Forward Direction:
    Index = 0, Element = Red
    Index = 1, Element = Green
```

```
Index = 2, Element = Yellow

Index = 3, Element = Orange

Index = 4, Element = Blue

Index = 5, Element = White



List Iterator in Backward Direction:

Index = 5, Element = White

Index = 4, Element = Blue

Index = 3, Element = Orange

Index = 2, Element = Yellow

Index = 1, Element = Green

Index = 0, Element = Red
```

Key Points:

1. If we use the next() method followed by the previous() method, the list iterator goes back to the same position.
2. The call to the next() method moves one index forward.
3. The call to the previous() method moves it one index backward.

How to Add and Set element in List while Java ListIterator iterating?

Java Collections Notes

Let's create a program where we will add an element into the list before the iterator position when Java ListIterator iterating list in the forward direction.

We will replace the last element returned by previous() method with the specified element during backward direction iteration.

Program source code 3:

```
import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;

public class AddDemo

{

public static void main(String[] args)

{

// Create an object of ArrayList of String type.

  List<String> list = new ArrayList<>();



// Adding elements to array list.

   list.add("A");

   list.add("B");

   list.add("C");

   list.add("D");

System.out.println("List: "+list);
```

```
System.out.println();


ListIterator<String> listIterator = list.listIterator();
System.out.println("Forward Direction Iteration:");
while(listIterator.hasNext())
{
  System.out.println(listIterator.next());
 }
listIterator.add("E"); // Adds an element before the iterator position.
System.out.println();
System.out.println(list);


System.out.println();


System.out.println("Backward Direction Iteration:");
while(listIterator.hasPrevious()){
 System.out.println(listIterator.previous());
}
listIterator.set("J"); // It will update the last element returned by previous.
```

```
System.out.println();

System.out.println(list);

 }

}
```

Output:

```
 List: [A, B, C, D]



Forward Direction Iteration:

A

B

C

D



[A, B, C, D, E]



Backward Direction Iteration:

E

D

C

B
```

```
A



[J, B, C, D, E]
```

How to Remove element from List while Java ListIterator iterating?

Let's take an example program where we will remove an element returned by the last call to next() method. Look at the program source code.

Program source code 4:

```
import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;



public class RemoveDemo

{

public static void main(String[] args)

{

 List<String> list = new ArrayList<>();

  list.add("A");

  list.add("B");
```

```
  list.add("C");
  list.add("D");

System.out.println("List: "+list);

ListIterator<String> listIterator = list.listIterator();
System.out.println("Forward Direction Iteration:");

while(listIterator.hasNext()){
  System.out.println(listIterator.next());
 }
listIterator.remove(); // Removes the last element returned by next method.
System.out.println("New List: " +list);
 }
}
```

Output:

List: [A, B, C, D]

Forward Direction Iteration:

A

B

```
C

D

New List: [A, B, C]
```

You need to be careful when calling the remove() method. It can be called only once after calling the next() or previous() method. If you call it immediately after a call to add() method, it throws an IllegalStateException.

Let's one more important example program where we will not use generic type. In this case, we will need to do type casting. We will also perform remove and add operations in this program. Look at the source code to understand better.

Program source code 5:

```
import java.util.ArrayList;

import java.util.ListIterator;

public class ListIteratorTest

{

 public static void main(String[] args)

 {

   ArrayList al = new ArrayList();

   al.add("Apple");

   al.add("Orange");

   al.add("Banana");

   al.add("Guava");
```

```
 al.add("Pineapple");
 System.out.println(al);


// Create the object of ListIterator using listIterator() method.
 ListIterator litr = al.listIterator();
 while(litr.hasNext())
 {
  Object o = litr.next();
  String str = (String)o; // Type casting.
 if(str.equals("Orange"))
 {
   litr.remove(); // It will remove orange from the list.
   System.out.println(al);
 }
 else if(str.equals("Guava"))
 {
   litr.add("Grapes"); // Adding Grapes after guava.
  System.out.println(al);
 }
 else if(str.equals("Pineapple"))
```

```
 {

   litr.set("Pears"); // Replacing Pineapple element.

   System.out.println(al);

  }

 }

}

}
```

Output:

```
    [Apple, Orange, Banana, Guava, Pineapple]

    [Apple, Banana, Guava, Pineapple]

    [Apple, Banana, Guava, Grapes, Pineapple]

    [Apple, Banana, Guava, Grapes, Pears]
```

Advantages of ListIterator in Java

ListIterator has several advantages. They are as follows:

1. List iterator in java supports many operations such as read, remove, replacement, and the addition of new objects.
2. Using the List Iterator, we can perform Iteration in both forward and backward directions.
3. Methods of ListIterator are easy and simple to use.

Limitations of ListIterator

ListIterator is the most powerful cursor but it still has some limitations. They are as follows:

1. Java List Iterator is applicable only for list implemented class objects. Therefore, it is not a universal Java cursor.
2. It is not applicable to whole collection API.

Similarities between Iterator and ListIterator

There are several similarities between Iterator and ListIterator cursors. They are as:

1. Both are introduced in Java 1.2 version.
2. Both are Iterators that are used to iterate elements of a collection object.
3. Both support read and delete operations.
4. Both support forward direction iteration.

Iterator	ListIterator
1. Java Iterator is applicable to the whole Collection API.	1. Java ListIterator is only applicable for List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc.
2. It is a Universal Iterator.	2. It is not a Universal Iterator in Java.
3. Iterator supports only forward direction Iteration.	3. ListIterator supports both forward and backward direction iterations.
4. It is known as a uni-directional iterator.	4. It is also known as bi-directional iterator.

5. Iterator supports only read and delete operations.	5. ListIterator supports all the operations such as read, remove, replacement, and the addition of the new elements.
6. We can get the Iterator object by calling iterator() method.	6. We can create ListIterator object by calling listIterator() method.

5. Both are not legacy interfaces.

Difference between Iterator and ListIterator

Now we will see the main differences between Iterator vs ListIterator in Java. They are as:

How to Iterate ArrayList in Java

In this tutorial, we will learn **how to iterate ArrayList in Java**. Basically, there are five ways by which we can iterate over elements of an ArrayList. That is, Java collections framework provides five ways to retrieve elements from a collection object. They are as follows:

1. Using for loop
2. Using Enhanced for loop or Advanced for loop
3. Using while Loop
4. By using Iterator
5. By ListIterator

Java Collections Notes

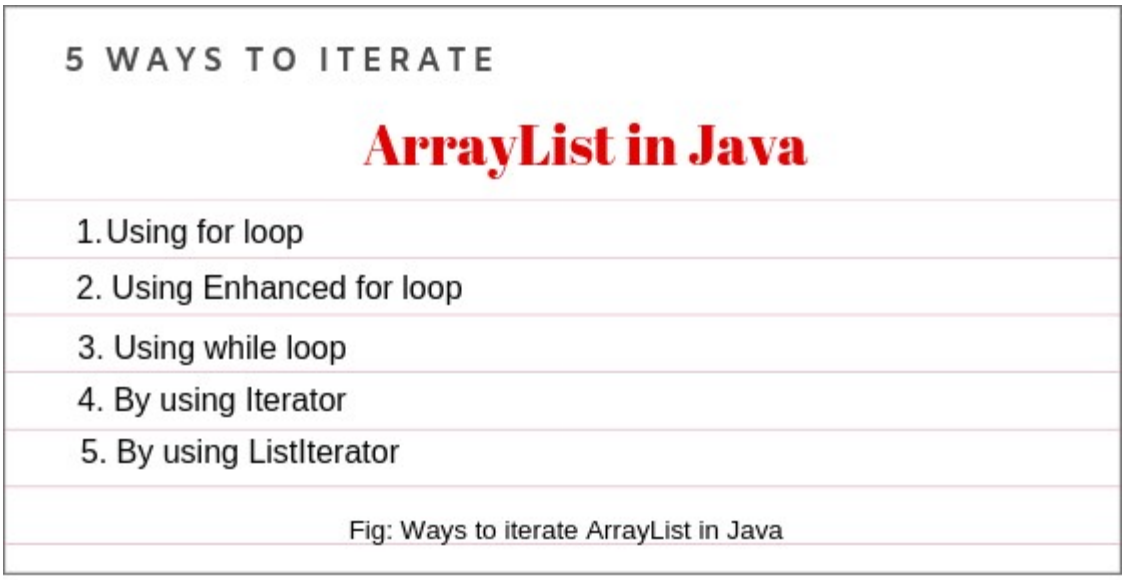


Fig: Ways to iterate ArrayList in Java

Iterate ArrayList in Java using for loop

The general syntax for simple for loop is as follows:

```
for(initialization; condition; step)

{

 - - body- -

 - - - - -- }
```

Where initialization is a variable declaration, condition is an expression of type boolean, step is an increment/decrement, and body is a statement.

The loop is executed as follows:
1. The initialization is executed.
2. The condition is evaluated if it is true, the body is executed. If it is false, the loop terminates.

For example:

```
// Get size of the ArrayList.
   int size = al.size();



// Iterate list of objects.
   System.out.println("for Loop");
   for(int i = 0; i < size; i++)
   {
// Call get() method to return or get the elements on the specified index after iterating.
     String getElement = al.get();
     System.out.println(getElement);
   }
```

Iterating ArrayList using Enhanced or Advanced for loop

This Enhanced for loop is introduced in Java 1.5 version. This Enhanced for loop is mostly used in the industry. We can easily use for-each loop to retrieve elements of a collection object.

There are three benefits of using for-each loop to iterate list of objects. They are as:

» In Enhanced for loop, we don't need to increment or decrement.
» No size calculation.
» We must use the generic concept to iterate elements of a list.

Java Collections Notes

The general syntax for Enhanced for loop is as follows:

```
for(data_type element : Object reference variable )

{

 - - - - - -

 - - - - - -

  body

 - - - - - - }
```

How Enhanced for loop works?

Enhance for loop works in two steps. They are as follows:

1. It iterates over each element in the collection or array.
2. It stores each element in a variable and executes the body.

Let's take some examples based on it.

```
1. ArrayList<String> al = new ArrayList<String>();

   for(String element: al)

   {

     System.out.println(element);

   }

2. ArrayList<Employee> al = new ArrayList<Employee>();

   for(Employee emp: al )

   {
```

```
    System.out.println(emp.name);

    System.out.println(emp.age);

  }
```

Let's take an example program where we will implement the concepts of simple for loop and enhanced for loop to understand better.

Program source code 1:

```
package iterateTest;

import java.util.ArrayList;

public class IterateArrayList

{

public static void main(String[] args)

{

// Create object of ArrayList of type String. In the list, we can add only String type of elements.

   ArrayList<String> al = new ArrayList<String>();



// Call add() method to add the elements in the list using reference variable al.

   al.add("A"); // Adding element at index 0.

   al.add("B"); // Adding element at index 1.

   al.add("C"); // Adding element at index 2.

   al.add("D"); // Adding element at index 3.
```

```
   al.add("E"); // Adding element at index 4.



// Displaying original elements of the ArrayList.

   System.out.println(al); // It will display all elements of ArrayList at a time.



// Iterating ArrayList using for loop and call size() method to get the size of elements.
Since the return type of size method is an integer. Therefore, we will store it using
variable elementsize of type int.

  System.out.println("Using for loop");

  int elementsize = al.size();

  System.out.println("Size: " +elementsize);

  for(int i = 0; i < al.size(); i++)

  {

// Call get() method to return elements on specified index after iterating.

   String getElement = al.get(i);

   System.out.println(getElement);

  }

 al.set(2, "G"); // It will replace current element at position 2 with element G.

 al.set(3, null); // adding null element at position 3.



// Iterating ArrayList using Enhance for loop.
```

```
  System.out.println("Using Enhance for loop");

  for(String element:al)

  {

    System.out.println(element);

   }

  }

}
```

Output:

```
    [A, B, C, D, E]

    Using for loop

    Size: 5

    A B C D E

    Using Enhance for loop

    A B G null E
```

Key point:

Iteration over elements in the ArrayList using Enhanced for loop is fail-fast. That means that we can not add or remove an element in the ArrayList during Iteration otherwise, it will throw **ConcurrentModificationException**.

Iterating ArrayList using While loop

The general syntax for while loop is as follows:

```
Initialization;

while(condition)

{

  statement1;

  statement2;

  Increment/decrement;

 }
```

Let's take an example based on it.

```
System.out.println("Iteration of ArrayList using while loop");

int i = 0; // initialization.

while(ar.size() > i)

{

  System.out.println(i);

   i++; // Increment.

}
```

Let's create a program where we will iterate elements of ArrayList using while loop concept.

Program source code 2:

```
package iterateTest;

import java.util.ArrayList;
```

```
public class IterateUsingWhilelloop

{

public static void main(String[] args)

{

 ArrayList<Integer> al = new ArrayList<Integer>();

  al.add(20);

  al.add(25);

  al.add(null);

  al.add(30);

  al.add(25);

 System.out.println(al);



// Iteration of ArrayList using while loop.

   System.out.println("Iteration using while loop");

   int i = 0;

  while(al.size() > i)

  {

    Integer itr = al.get(i);

    System.out.println(itr);

     i++;
```

```
  }

 }

}
```

Output:

```
    [20, 25, null, 30, 25]

    Iteration using while loop

    20 25 null 30 25
```

How to iterate ArrayList in Java using Iterator

The iterator() method is used to iterate over elements of ArrayList in Java. It is useful when we want to remove the last element during iteration. The Iterator interface defines three methods. They are as follows:

1. hasNext(): This method returns true if the iteration has more elements in the forward direction.
2. next(): It returns the next element in the iteration. If the iteration has no more elements then it will throw "NoSuchElementException".
3. remove(): The remove() method removes the last element returned by the iterator. This method must be called after calling next() method otherwise, it will throw "IllegalStateException".
For more detail with diagram, go to this tutorial: Iterator in Java | Methods, Iterable Interface, Example
Let's make a program where we will iterate elements of ArrayList by using iterator. We will also remove the last element returned by the iterator.

Program source code 3:

```
package iterateTest;
```

```
import java.util.ArrayList;

import java.util.Iterator;

public class RemoveTest

{

public static void main(String[] args)

{

 ArrayList<String> al = new ArrayList<String>();

  al.add("Apple");

  al.add("Mango");

  al.add("Banana");

  al.add("Guava");

  al.add("Pineapple");

System.out.println(al); // It will print all elements at a time.



   System.out.println("Iteration using iterator concept.");

// Create an object of Iterator by calling iterator() method using reference variable al.

// At the beginning, itr(cursor) will point to index just before the first element in al.

   Iterator<String> itr = al.iterator();



// Checking the next element availability using reference variable itr.
```

```
  while(itr.hasNext())
  {
// Moving cursor to next element using reference variable itr.
    String str = itr.next();
    System.out.println(str);

// Removing the pineapple element.
  if(str.equals("Pineapple"))
  {
    itr.remove();
    System.out.println("After removing pineapple element");
    System.out.println(al);
   }
  }
 }
}
```

Output:

```
    [Apple, Mango, Banana, Guava, Pineapple]
    Iteration using iterator concept.
    Apple Mango Banana Guava Pineapple
```

```
After removing pineapple element

[Apple, Mango, Banana, Guava]
```

Key point:

iterator returned by ArrayList is a fail-fast. That means that if we add an element or remove an element in the ArrayList during the Iteration then it will throw "ConcurrentModificationException".

This is because the loop internally creates a fail-fast iterator which throws an exception whenever any structural modification in the underlying data structure.

Let's make a program where we will understand the concept of fail-fast during the iteration of elements.

Program source code 4:

```
package iterateTest;

import java.util.ArrayList;

import java.util.Iterator;

public class AddingElementUsingIteration

{

public static void main(String[] args)

{

 ArrayList<String> al = new ArrayList<String>();

  al.add("Lion");

  al.add("Tiger");

  al.add("Elephant");
```

```
  al.add("Bear");

Iterator<String> itr = al.iterator();

while(itr.hasNext())

{

  System.out.println(itr.next()); // Adding element during iteration. Since the return type of add() method is boolean. Therefore, we will store it using variable b with data type boolean.

  boolean b = al.add("Leopard"); // Compile time error. It will throw ConcurrentModificationException.

  System.out.println(b);

  }

 }

}

Output:

      Exception in thread "main" java.util.ConcurrentModificationException
```

How to iterate List of Objects in Java using ListIterator

In Java programming, ListIterator is used to iterate the elements of array list in both forward as well as backward directions. Using ListIterator, we can also start the iteration from a specific element in the ArrayList.

We can perform different kinds of operations such as read, remove, replacement (current object), and the addition of the new objects.

ListIterator interface defines 9 methods in Java. Since ListIterator is the child interface of Iterator. Therefore, all the methods of Iterator are available by default to ListIterator. ListIterator interface has total of 9 methods.

For more detail with diagram, go to ListIterator tutorial: ListIterator in Java

Let's make a program where we will iterate elements of ArrayList in both forward and backward directions using ListIterator.

Program source code 5:

```
package iterateTest;

import java.util.ArrayList;

import java.util.ListIterator;

public class ArrayListUsingListIterator

{

public static void main(String[] args)

{

 ArrayList al = new ArrayList();

  al.add("First");

  al.add("Second");

  al.add("Third");

  al.add("Fourth");

  al.add("Fifth");

System.out.println(al);
```

```
// Iterating using ListIterator.

// Call listIterator() method to create object of ListIterator using reference variable al.

   ListIterator litr = al.listIterator(); // Here, we are not using generic. Therefore, Typecasting is required.



// Checking the next element availability in the forward direction using reference variable litr.

    System.out.println("Iteration in the forward direction");

  while(litr.hasNext())

  {

// Moving cursor to next element in the forward direction using reference variable litr.

    Object o = litr.next();

    String str = (String)o; // Typecasting is required because the return type of next() method is an Object.

    System.out.println(str);

 }

// Checking the previous element in the backward direction using reference variable litr1.

    System.out.println("Iteration in the backward direction.");

  while(litr.hasPrevious())

  {
```

```
// Moving cursor to the previous element in the backward direction.

   Object o = litr.previous();

   String str1 = (String)o; // Typecasting.

   System.out.println(str1);

  }

 }

}
```

```
Output:

      [First, Second, Third, Fourth, Fifth]

      Iteration in the forward direction

      First Second Third Fourth Fifth

      Iteration in the backward direction.

      Fifth Fourth Third Second First
```

Let's make a program, we will perform add and remove operations using ListIterator. We will use generic in this program so that we do not require Typecasting.

Program source code 6:

```
package iterateTest;

import java.util.ArrayList;

import java.util.ListIterator;

public class AddRemoveTest
```

```
{
public static void main(String[] args)
{
 ArrayList<String> al = new ArrayList<String>();
  al.add("One");
  al.add("Two");
  al.add("Three");
  al.add("Nine");
  al.add("Five");
  al.add("Seven");
 System.out.println(al);

 ListIterator<String> litr = al.listIterator();
 while(litr.hasNext())
 {
  String str = litr.next();
  if(str.equals("Nine"))
  {
    litr.remove();
    litr.add("Four");
```

```
    System.out.println(al);

   }

 else if(str.equals("Seven"))

 {

   litr.set("Six");

   System.out.println(al);

  }

 }

 }

}
```

Output:

```
      [One, Two, Three, Nine, Five, Seven]

      [One, Two, Three, Four, Five, Seven]

      [One, Two, Three, Four, Five, Six]
```

Let's create one more program where we will add elements from 0 to 9 numbers in the list by using "for loop". But we will iterate from a specific element '4' in the list.

Program source code 7:

```
package iterateTest;

import java.util.ArrayList;

import java.util.ListIterator;
```

```
public class SpecificElementTest

{

public static void main(String[] args)

{

  ArrayList<Integer> al = new ArrayList<Integer>();

  for(int i = 0 ; i < = 9 ; i++)

  {

     al.add(i);

  }

 System.out.println(al);

 ListIterator<Integer> litr = al.listIterator(4); // Iterating through a specific element '4'.

 while(litr.hasNext())

 {

   Integer it = litr.next();

   System.out.println(it);

  }

 while(litr.hasPrevious())

 {

   al.add(20); // It will throw ConcurrentModificationException because we can not add
element in the ArrayList during Iteration.
```

```
  Integer it1 = litr.next();

  System.out.println(it1);

 }

}

}

Output:

     [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

     4 5 6 7 8 9

     Exception in thread "main" java.util.ConcurrentModificationException
```

Key point:
ListIterator returned by ArrayList is also fail-fast. It will throw ConcurrentModificationException. That is, we cannot add or remove an element in the ArrayList during Iteration.

How to Synchronize ArrayList in Java

In this tutorial, we will learn how to synchronize ArrayList in Java. Before going to understand this topic, first, we will understand in brief the meaning of synchronization.

What is Synchronization in Java?

Technically, Synchronization in java is the process of allowing only one thread at a time to complete the task entirely. It allows only one thread to access a single resource or a single task.

Java Collections Notes

When multiple threads access the same resources at the same time then the program may produce an undesirable output. So, this problem can be solved by using a technique known as synchronization. The objective of synchronization is to control access to a shared resource.

We know that by default ArrayList class is not a thread-safe or non-synchronized. That means the multiple threads can access the same ArrayList object or instance simultaneously.

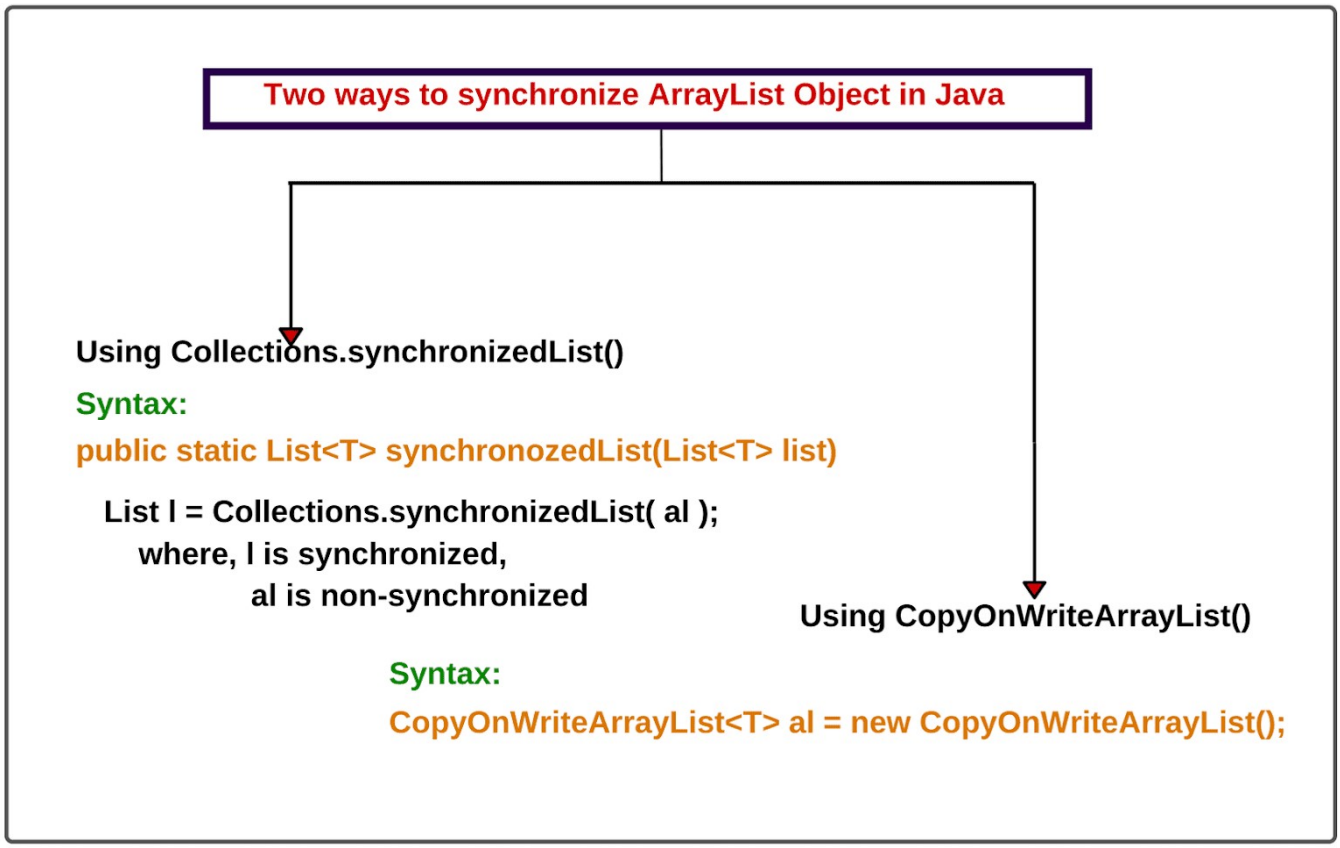
Therefore, it cannot be used in the multi-threading environment without explicit synchronization. This is because if ArrayList is used in the multi-threading environment without using the synchronization technique then it may produce unpredictable output.

So, how will we get synchronized ArrayList in Java with thread safety?

How to Synchronize ArrayList object in Java?

There are two methods by which we can get the synchronized version of ArrayList. They are as follows:

1. By using Collections.synchronizedList() method
2. By using CopyOnWriteArrayList



Synchronization of ArrayList using Collections.synchronizedList() method

---

This method is used to synchronize collections in Java. The syntax of this method is as follows:

Syntax:

```
public static List<T> synchronizedList(List<T> list)
```

1. The synchronizedList() method accepts List which could be the implementation of List interface. For example, ArrayList, and LinkedList.

2. The return type of this method is a synchronized list (thread-safe).

3. synchronizedList is the name of the method.

4. The parameter list is the list to be wrapped in the synchronize list.

5. T represents the type of generic.

Note: The Iterator is used in the synchronization block for synchronization to avoid ConcurrentModificationException in Java. The iterator returned by the synchronized ArrayList is fail-fast. It will throw ConcurrentModificationException when any modification happens in the list during the iteration.
Let us take a basic example to synchronize ArrayList in java.

```
ArrayList<String> al = new ArrayList<String>(); // Non-synchronized.

 List l = Collections.synchronizedList( al );

       l ➞  synchronized.

       al ➞ Non-synchronized.

or,

 List<String> synlist = Collections.synchronizedList( new ArrayList<String>);
```

Similarly, we can get the synchronized version of Set, Map objects by using the following syntax:

```
public static Set synchronizedList( Set s );

public static Map synchronizedList( Map m );
```

Let's take an example program where we will synchronize the list of non-synchronized ArrayList objects and then we will call iterator() method to iterate the list of synchronized ArrayList objects.

Program source code 1:

```
package synchronizeTest;
```

Java Collections Notes

```
import java.util.ArrayList;

import java.util.Collections;

import java.util.Iterator;

import java.util.List;



public class ArrayListSynchronizationTest

{

public static void main(String[] args)

{

// Create an ArrayList object with initial capacity of 10.

// Non-synchronized ArrayList Object.

   List<String> l = new ArrayList<String>();



// Add elements in the list.

    l.add("Apple");

    l.add("Orange");

    l.add("Banana");

    l.add("Pineapple");

    l.add("Guava");
```

```
// Synchronizing ArrayList in Java.

   List<String> synlist = Collections.synchronizedList( l ); // l is non-synchronized.



// Here, we will use a synchronized block to avoid the non-deterministic behavior.

   synchronized(synlist)

   {

// Call iterator() method to iterate the ArrayList.

    Iterator<String> itr = synlist.iterator();

    while(itr.hasNext())

    {

      String str = itr.next();

      System.out.println(str);

    }

  }

 }

}
```

Output:

Apple Orange Banana Pineapple Guava

In the preceding program, ArrayList achieves thread-safety by calling the synchronizedList() method that locks the whole list into the synchronized block.

Let's take another program where we will check the occurrence of ConcurrentModificationException by adding a number into the synchronized list during the Iteration.

Program source code 2:

```
package synchronizeTest;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Iterator;

import java.util.List;

public class CMETest

{

public static void main(String[] args)

{

  ArrayList<Integer> al = new ArrayList<Integer>();

  for(int i = 0; i < 10; i++)

  {

    al.add(i);

  }

 List<Integer> synlist = Collections.synchronizedList(al);
```

```
 synchronized(synlist)

 {

  Iterator<Integer> itr = synlist.iterator();

  while(itr.hasNext())

  {

    al.add(20); // It will throw ConcurrentModificationException because we cannot
modify list during Iteration.

    System.out.println(itr.next());

  }

  }

 }

}

Output:

     Exception in thread "main" java.util.ConcurrentModificationException
```

Synchronization of ArrayList using CopyOnWriteArrayList class in Java

---

1. CopyOnWriteArrayList implements List interface and creates an empty list.
2. It creates a list of the elements in the order of specified collection.
3. It is thread-safe concurrent access of ArrayList. When ArrayList is modified, it will create a fresh copy of the underlying array.

4. Iterator and ListIterator returned by CopyOnWriteArrayList is completely thread-safe. It will not throw ConcurrentModificationException when ArrayList is modified during iteration.
5. CopyOnWriteArrayList does not lock the whole list. When a thread writes into the list, It simply replaces the list by a fresh copy of the underlying array.

In this way, it provides concurrent access of ArrayList for the multiple threads without locking. Since read operation is a thread-safe, two threads cannot write into the list simultaneously.

The syntax to create an object of CopyOnWriteArrayList is as follows:

Syntax:

```
CopyOnWriteArrayList<T> al = new CopyOnWriteArrayList<T>(); // T is generic.
```

Let's create a program where we will synchronize a list of elements in ArrayList using the CopyOnWriteArrayList class.

Program source code 3:

```
package synchronizeTest;

import java.util.Iterator;

import java.util.concurrent.CopyOnWriteArrayList;

public class SynArrayListUsingCOWA

{

public static void main(String[] args)

{

// Create a thread-safe ArrayList.

  CopyOnWriteArrayList<String> al = new CopyOnWriteArrayList<String>();
```

```
   al.add("Pen");

   al.add("Pencil");

   al.add("Copy");

   al.add("Eraser");

   al.add("Shapner");

  System.out.println("Displaying synchronized ArrayList ");



// Synchronized block is not required in this method.

  Iterator<String> itr = al.iterator();

  while(itr.hasNext())

  {

    String str = itr.next();

    System.out.println(str);

  }

 }

}
```

Output:

```
     Displaying synchronized ArrayList

     Pen Pencil Copy Eraser Shapner
```

Let's make another program where we will try to add an element in the synchronized list during iteration. Here, we will check that CopyOnWriteArrayList will throw ConcurrentModificationException during the iteration or not.

Program source code 4:

```
package synchronizeTest;

import java.util.Collections;

import java.util.Iterator;

import java.util.List;

import java.util.concurrent.CopyOnWriteArrayList;



public class AddingTest

{

public static void main(String[] args)

{

 CopyOnWriteArrayList<String> al = new CopyOnWriteArrayList<String>();

   al.add("A");

   al.add("B");

   al.add(null);

   al.add("D");

   al.add("E");

   al.add("H");
```

```
 System.out.println(al);

 List<String> synlist = Collections.synchronizedList(al);



// Here, Synchronized block is not required.

// Call iterator() method using reference variable synlist.

   Iterator<String> itr = synlist.iterator();

   while(itr.hasNext())

    {

     al.set(5, "F"); // It will not throw ConcurrentModificationException during Iteration.

     String str = itr.next();

     System.out.println(str);

    }

   System.out.println(al);

  }

 }
```

Output:

```
      [A, B, null, D, E, H]

      A B null D E H

      [A, B, null, D, E, F]
```

As you can observe in the above program, during iteration, we can easily add an element in the synchronized list using the CopyOnWriteArrayList method.

ArrayList Program in Java for Interview Practice

In this tutorial, we have listed the various types of ArrayList programs in Java for practice.

All example programs with explanations are important for beginners to understand the concepts of ArrayList in Java.

I will recommend that before going to practice all these programs, you must read the previous tutorial of ArrayList in Java that is very helpful for interviews.

How to add elements to ArrayList in Java Dynamically?

Let's take an example program where we will add elements in the array list dynamically. Look at the source code.

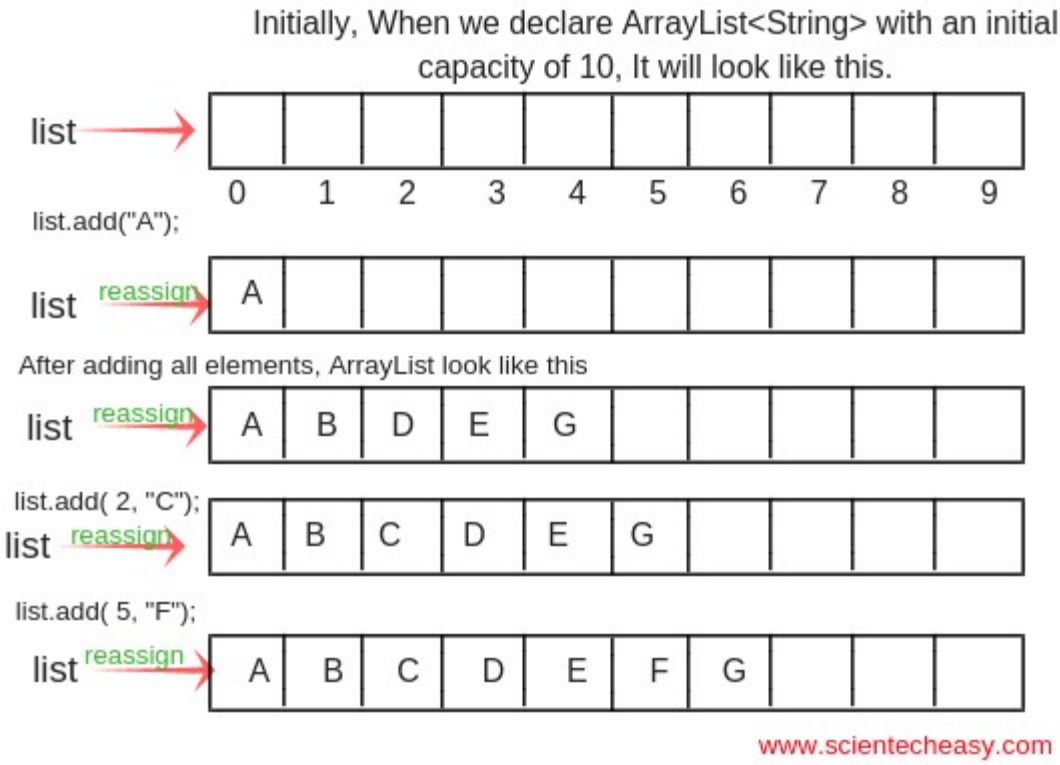
Program source code 1:

```
package arrayListProgram;

import java.util.ArrayList;

import java.util.List;

public class ArrayListEx1

{

public static void main(String[] args)

{

// Creates an ArrayList object with an initial capacity of 10.

   List<String> list = new ArrayList<String>();
```

```
// Call add() method to add elements at the end of the list using the reference variable list.
    System.out.println("Adding elements to end of list");


    list.add("A"); // Adding elemen at index 0.
    list.add("B"); // Adding element at index 1.
    list.add("D"); // Adding element at index 2.
    list.add("E"); // Adding element at index 3.
    list.add("G"); // Adding element at index 4.


System.out.println("ArrayList insertion order: " +list);


System.out.println("Adding an element at a specific index after B element.");
   list.add(2, "C");
System.out.println("ArrayList insertion order after adding C: " +list );


System.out.println("Adding an element at a specific index after E");
   list.add(5, "F");
System.out.println("ArrayList insertion order after adding F: " +list);
 }
}
```

```
Output:

    Adding elements to the end of the list

    ArrayList insertion order: [A, B, D, E, G]

    Adding an element at a specific index after B element.

    ArrayList insertion order after adding C: [A, B, C, D, E, G]

    Adding an element at a specific index after E

    ArrayList insertion order after adding F: [A, B, C, D, E, F, G]
```

Explanation with diagram:



In this example program, when we added a new element 'C' at a specific index 2, elements D, E, and G are shifted to the one position left as shown in the above figure.

After inserting element C, the reference variable 'list' will reassign to the new array list and copy all the elements into the new ArrayList. The old default array list will be gone to the garbage collection.

Now the position of element E is shifted from position 3 to position 4 in the new ArrayList. Therefore, the element F after E has been added at position 5, not at position 4.

Program source code 2:

```
package arrayListProgram;

import java.util.ArrayList;

public class ArrayListEx2

{

public static void main(String[] args)

{

// Creates an ArrayList1 with an initial capacity of 10.

   ArrayList<String> al = new ArrayList<String>();

    al.add("G");

    al.add("H");

    al.add("I");

    al.add("M");

    al.add("N");

  System.out.println("Original ArrayList insertion order: " +al);
```

```
// Creates another ArrayList2 with an initial capacity of 10.
   ArrayList<String> al2 = new ArrayList<String>();
    al2.add("J");
    al2.add("K");
    al2.add("L");

// Call addAll() method to add all elements in the list2 at spcefic index 3.
   al.addAll(3, al2);
   System.out.println("ArrayList insertion order after adding group of elements in the list1
: " +al);

// Creates ArrayList3 with initial capacity 10.
   ArrayList<String> al3 = new ArrayList<String>();
    al3.add("20");
    al3.add("40");
    al3.add("12");

// Adding group of elements at the end of list1.
    al.addAll(al3);
```

```
   System.out.println(al);

  }

}
```

Output:

```
      Original ArrayList insertion order: [G, H, I, M, N]

      ArrayList insertion order after adding a group of elements in the list1 : [G, H, I, J, K, L, M, N]

      [G, H, I, J, K, L, M, N, 20, 40, 12]
```

Java ArrayList Program based on Remove, Get, Contains, and Set methods

Let's create a program where we will perform the following operations such as remove, get, contains, and set on elements of the list. Look at the source code.

Program source code 3:

```
package arrayListProgram;

import java.util.ArrayList;

public class ArrayListEx3

{

public static void main(String[] args)

{

  ArrayList<String> al = new ArrayList<String>();

   al.add("Science");

   al.add("Maths");
```

```
   al.add("Hindi");

   al.add("English");

   al.add("Social Science");

 System.out.println("Original ArrayList insertion order: " +al);



// Call get() method to get element at the index 2.

   String str = al.get(2); // Since the return type of get method is String. So we will store it using str variable with data type string.

   System.out.println("Element at index 2: " +str);



// Call contains() method to check whether the element English is present in the list or not.

   boolean b = al.contains("English"); // Since the return type of contains method is boolean. So we will store it by using 'b' variable with type boolean.



// The contains() method returns true if the specified element is present in the list otherwise false.

   System.out.println(b); // return true.

   boolean b2 = al.contains("Sanskrit");

   System.out.println(b2); // return false.
```

```
   boolean bo=al.containsAll(al); // It will return truse if this collection contains all elements in the specified collection.

   System.out.println(bo);



// call remove() method to remove element English at a specified index.

   al.remove(3);

   System.out.println("After removing element, ArrayList Order: " +al);



// Call set() method to replace element Social Science.

   al.set(3, "Computer");

   System.out.println("After replacing element, ArrayList Order: " +al);

  }

}
```

Output:

```
    Original ArrayList insertion order: [Science, Maths, Hindi, English, Social Science]

    Element at index 2: Hindi

    true

    false

    true

    After removing element, ArrayList Order: [Science, Maths, Hindi, Social Science]
```

```
After replacing element, ArrayList Order: [Science, Maths, Hindi, Computer]
```

Iteration ArrayList Program using Iterator, ListIterator, Enumeration, & Enhanced for loop

Let's make a program where we will iterate elements of array list using Iterator, ListIterator, Enumeration, and for loop.

Program source code 4:

```
package arrayListProgram;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Enumeration;

import java.util.Iterator;

import java.util.List;

import java.util.ListIterator;



public class ArrayListEx4

{

public static void main(String[] args)

{

 List<Integer> list = new ArrayList<Integer>();

  list.add(20);

  list.add(25);
```

```
  list.add(30);

  list.add(35);

  list.add(40);

System.out.println("Original ArrayList: " +list);

System.out.println("--Using Iterator--");



// Call iterator() method to iterate over the elements.

   Iterator<Integer> itr = list.iterator();



// Checking the next element availability using reference variable itr.

   while(itr.hasNext())

   {

// Moving the cursor to next element using reference variable itr.

     Integer it = itr.next(); // Here, return type is an integer.

     System.out.println(it);

   }

System.out.println("--Iterating in forwarding direction using ListIterator--");



// Call ListIterator() method to iterate over the elements.

   ListIterator<Integer> litr = list.listIterator();
```

```
// Checking the next element availability using reference variable litr.
   while(litr.hasNext())
   {
     Integer lit = litr.next();
     System.out.println(lit);
    }
System.out.println("--Using Enumeration-- ");

Enumeration<Integer> enumlist = Collections.enumeration(list);
while(enumlist.hasMoreElements())
 {
   System.out.println(enumlist.nextElement());
  }
System.out.println("--Using Enhanced for loop--");
for(Integer in : list)
{
   System.out.println(in);
  }
 }
```

```
 }

}
```

Output:

```
      Original ArrayList: [20, 25, 30, 35, 40]

      --Using Iterator--

      20 25 30 35 40

     --Iterating in forwarding direction using ListIterator--

      20 25 30 35 40

     --Using Enumeration--

      20 25 30 35 40

     --Using Enhanced for loop--

      20 25 30 35 40
```

ArrayList Program based on isEmpty, Size, and Clear methods

Let's create a program where we will perform the following operations such as checking of elements in the list, size of list, and delete all elements from the list. To perform these operations n the program, we will call three methods isEmpty(), size(), and clear() methods.

Program source code 5:

```
package arrayListProgram;

import java.util.ArrayList;

import java.util.List;

public class ArrayListEx5
```

```
{

public static void main(String[] args)

{

// Creates an array with an initial capacity of 10.

   List<String> list = new ArrayList<String>();



// Call isEmpty() method to check elements in the list.

   boolean b = list.isEmpty(); // It will return true if this list contains no elements.

   System.out.println(b);



// Adding elements to the end of the list.

   list.add("INDIA");

   list.add("USA");

   list.add("SRILANKA");

   list.add("JAPAN");

   list.add("FRANCE");

 System.out.println("Original ArrayList insertion order: " +list);



// Call size() method to get the number of elements or length of ArrayList in the list.

   Integer sizelist = list.size();
```

```
System.out.println("Size/Length of the list: " +sizelist);

System.out.println(list.isEmpty()); // Return false.



System.out.println("--Clear all elements from the list--");

 list.clear();

boolean bol = list.isEmpty();

System.out.println(bol);



Integer size = list.size();

System.out.println("No of elements in the list: " +size);

 }

}

Output:

      true

      Original ArrayList insertion order: [INDIA, USA, SRILANKA, JAPAN, FRANCE]

      Size/Length of the list: 5

      false

     --Clear all elements from the list--

      true

      No of elements in the list: 0
```

Program on ArrayList based on getting Index of First occurrence of Element

Let's create a program where we will get the first occurrence of element from the list.

Program source code 6:

```
package arrayListProgram;

import java.util.ArrayList;

public class ArrayListEx6

{

public static void main(String[] args)

{

 ArrayList<String> al = new ArrayList<String>();

  al.add("ABC");

  al.add("DEF");

  al.add("ABC");

  al.add("GHI");

  al.add(null); // null element is allowed in the ArrayList.

System.out.println("Original ArrayList order:" +al);



// Call indexOf() method to get the index of the first occurrence of a specific element in the list.

  Integer indexofElement = al.indexOf("ABC"); // It will return the index of the first occurrence of the element.
```

```
  System.out.println("Index of the element ABC: " +indexofElement);



Integer indexElement = al.indexOf(null);

System.out.println("Index of the element null: " +indexElement);

Integer indexE = al.indexOf("JKL"); // It will return -1 if the element is not present in the list.

System.out.println("Index of the element JKL: " +indexE);

 }

}

Output:

      Original ArrayList order:[ABC, DEF, ABC, GHI, null]

      Index of the element ABC: 0

      Index of the element null: 4

      Index of the element JKL: -1
```

ArrayList Program based on getting Index of Last occurrence of Element

Program source code 7:

```
package arraylistProgram;

import java.util.ArrayList;

public class ArrayListEx7

{

public static void main(String[] args)
```

```
{
 ArrayList<Integer> arlist = new ArrayList<Integer>();
  arlist.add(20);
  arlist.add(25);
  arlist.add(30);
  arlist.add(35);
  arlist.add(40);
  arlist.add(25);
  arlist.add(20);
 System.out.println("Original ArrayList Order: " +arlist);


Integer lastindex = arlist.lastIndexOf(25);
System.out.println("Index of last occurrence of the element 25: " +lastindex);
Integer lindex = arlist.lastIndexOf(20);
System.out.println("Index of the element 20: " +lindex );
  }
}
Output:
      Original ArrayList Order: [20, 25, 30, 35, 40, 25, 20]
      Index of last occurrence of the element 25: 5
```

```
Index of the element 20: 6
```

8. Assume that list1 is an array list that contains strings Apple, Mango, and Grapes, and list2 is another array list that contains strings Apple, Mango, and Guava.
Answer the following questions:

a. What are list1 and list2 after executing list1.addAll(list2)?
b. What are list1 and list2 after executing list1.add(list2)?
c. What are list1 and list2 after executing list1.removeAll(list2)?
d. What are list1 and list2 after executing list1.remove(list2)?
e. What are list1 and list2 after executing list1.retainAll(list2)?
f. What is list1 after executing list1.clear()?

Answer:

a) List1: [Apple, Mango, Grapes, Apple, Mango, Guava]

List2: [Apple, Mango, Guava]

b) Compile time error: The method add(String) in the type List<String> is not applicable for the arguments (List<String>).

c) List1: [Grapes]

List2: [Apple, Mango, Guava]

d) List1: [Apple, Mango, Grapes]

List2: [Apple, Mango, Guava]

e) List1: [Apple, Mango]

List2: [Apple, Mango, Guava]

f) List1: [ ]

9. What is wrong with each of the following code?

```
a. ArrayList<int> al = new ArrayList<int>();

b. ArrayList<String> al = new ArrayList();

c. ArrayList<String> al = new ArrayList<String>;
```

```
d. ArrayList<Integer> al = new ArrayList<Integer>();
   for (int i = 1; i <= 10; i++)
   {
     al.set(i - 1, i * i);
   }
e. ArrayList<Integer> al;
   for (int i = 1; i <= 10; i++)
   {
     al.add(i * i);
   }
```

Answer:

a) Syntax error.
b) No problem.
c) Syntax error.
d) IndexOutOfBoundsException
e) Compile time error: The local variable al may not have been initialized.

10. What is the output of the following program?

```
public class AddDemo
{
public static void main(String[] args)
{
 ArrayList<String> al = new ArrayList<String>();
```

```
  al.add("A");

  al.add(0, "B");

  al.add("C");

  al.remove(1);

System.out.println(al);

 }

}
```

Output: [B, C]

Realtime Use of ArrayList in Java with Example

We have familiarized that an ArrayList is the most basic type of collection in Java. It uses an array internally to store data when we add elements to the array list.

The ArrayList class automatically manages the size of the array when we add an element to the array list.

When the underlying array is fixed, ArrayList class automatically creates a new array with a larger size or capacity and copies all the existing elements to the new array before adding the new elements.

The old array of the collection will be gone in the garbage collection. Now, we will understand realtime use of ArrayList in Java with example program. That is, when to use ArrayList in Java application.

An ArrayList can be used in Java Application when

1. We want to add duplicate elements to the list. Since Java ArrayList class allows duplicate elements, we can add duplicate elements to the list.

Java Collections Notes

Let's take an example program where we will use ArrayList to add duplicate elements in the list.

Program source code 1:

```
package arrayListProgram;

import java.util.ArrayList;

import java.util.Iterator;

public class DuplicateElementsEx

{

public static void main(String[] args)

{

// Create an ArrayList with an initial capacity of 10.

   ArrayList<String> al = new ArrayList<String>();



// Adding elements to the list.

    al.add("A"); // Adding element at index 0.

    al.add("B"); // Adding element at index 1.

    al.add("C"); // Adding element at index 2.

    al.add("D"); // Adding element at index 3.

   System.out.println("Original insertion array list order: " +al);



// Adding a duplicate element at position 2 and end of the list.
```

```
   al.add(2, "B");

   al.add("A");

   System.out.println("After adding duplicate element, ArrayList insertion order ");



// Call iterator() method to iterate over the elements of the array list.

   Iterator<String> itr = al.iterator();

   while(itr.hasNext())

   {

     String str = itr.next();

     System.out.println(str);

   }

  }

}

Output:

      Original insertion array list order: [A, B, C, D]

      After adding duplicate element, ArrayList insertion order

      A B B C D A
```

2. The size of ArrayList is growable in nature. Suppose we have a series of elements in a program that we need to handle on a dynamic basis.

Sometimes, this series of elements can be 5, or sometimes, it can be 3 or 8. i.e size is growable. So, in this case, you can use ArrayList.

Let's take an example program based on the growable nature of ArrayList.

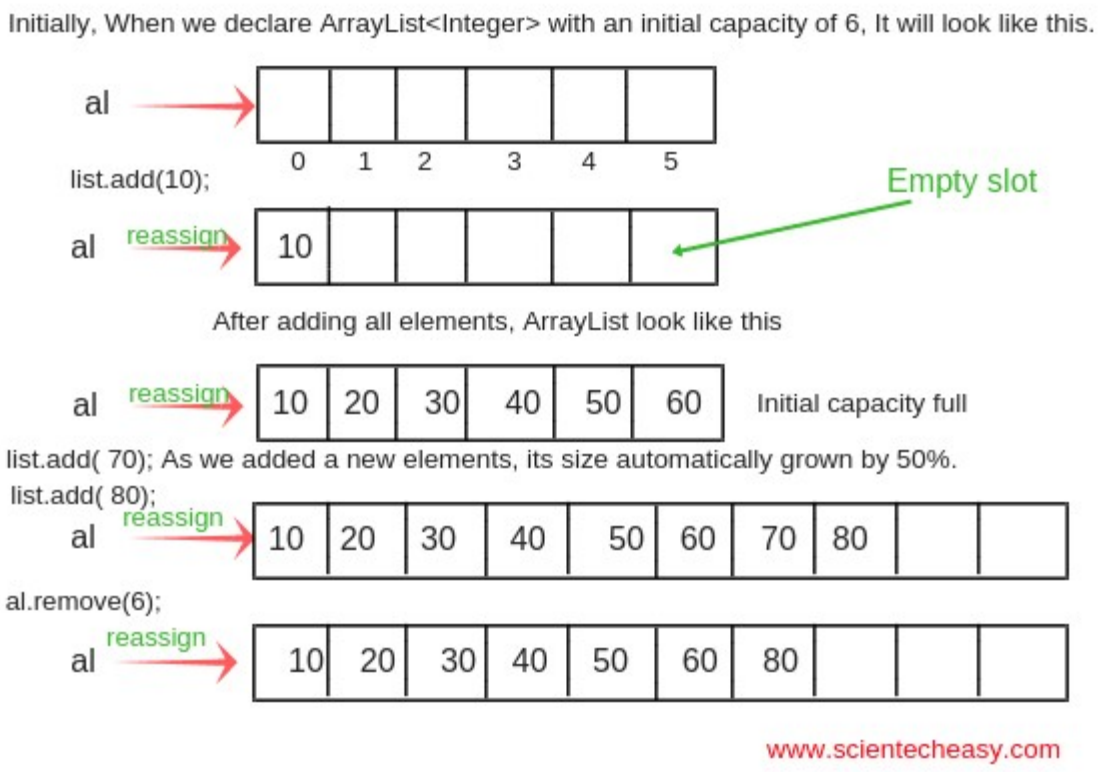
Program source code 2:

```
package com.arraylisttest;

import java.util.ArrayList;

public class GrowableArrayListEx

{

public static void main(String[] args)

{

// Create an array list with an initial capacity of 6.

   ArrayList<Integer> al = new ArrayList<Integer>(6);

    al.add(10);

    al.add(20);

    al.add(30);

    al.add(40);

    al.add(50);

    al.add(60);

  System.out.println(al);
```

```
 Integer sizelist = al.size();

 System.out.println("Original size: " +sizelist);



// Since the initial capacity of array list is full. Therefore, when we add a new element to the list, its capacity is automatically grown by 50%.

    al.add(70);

    al.add(80);

    Integer size = al.size();

    System.out.println("After adding new elements, ArrayList's size: " +size);



    al.remove(6);

    System.out.println(al);

   }

}
```

Explanation:

Look at the below figure to understand better how ArrayList grows its capacity internally.




```
Output:

    [10, 20, 30, 40, 50 60]

    Original size: 6

    After adding new elements, ArrayList's size: 8

    [10, 20, 30, 40, 50, 60 80]
```

3. We want to store null elements on the list. We can add any number of null elements in ArrayList.

Let's take an example program where we will add null elements in the list using ArrayList. Look at the source code.

Program source code 3:

```
package arrayListProgram;
```

```
import java.util.ArrayList;

import java.util.Iterator;

public class NullTestEx

{

public static void main(String[] args)

{

// Creates an ArrayList with an initial capacity of 10.

 ArrayList<String> arlist = new ArrayList<String>();



 arlist.add("Ganga");

 arlist.add("Yamuna");

 arlist.add(null); // Adding null element.

 arlist.add("Godavari");

 arlist.add("Ganga"); // Adding duplicate element.

 arlist.add(null); // Adding null element.

System.out.println("Original order: " +arlist);



Iterator<String> itr = arlist.iterator();

 while(itr.hasNext())

 {
```

```
   System.out.println(itr.next());

  }

}
```

Output:

```
     Original order: [Ganga, Yamuna, null, Godavari, Ganga, null]

     Ganga Yamuna null Godavari Ganga null
```

4. Getting elements from the list is more as compared to adding and removing elements. In this case, ArrayList is more preferred as compared to LinkedList because ArrayList implements a random access interface.

The retrieval of elements is very slow in LinkedList due to traverse from the beginning or ending to reach element.

Program source code 4:

```
package arrayListProgram;

import java.util.ArrayList;

public class GettingElementEx

{

public static void main(String[] args)

{

  ArrayList<String> al = new ArrayList<String>();

   al.add("a");

   al.add("ab");

   al.add("abc");
```

```
  al.add("abcd");
  al.add("a");
 System.out.println(al);


// Call get() method using reference variable al to get elements.
  System.out.println("Getting element from position 0 to 4 ");
  for(int i = 0; i < = 4 ; i++)
  {
    String getElement = al.get(i);
    System.out.println(getElement);
  }
 String getE = al.get(3); // Randomly getting element from position 3.
 System.out.println("Getting element at position 3: " +getE);
 }
}
Output:
      [a, ab, abc, abcd, a]
      Getting element from position 0 to 4
      a ab abc abcd a
      Getting element at position 3: abcd
```

5. We are not working in the multithreading environment in Java applications because ArrayList is not synchronized. Due to which multiple threads can access the same ArrayList object simultaneously.

To make ArrayList as synchronized, we will have to use Collections.synchronizedList() method.

Program source code 5:

```
package arrayListProgram;

import java.util.ArrayList;

public class MultiThreadEx extends Thread

{

// Declare ArrayList variable.

   ArrayList<Integer> list;



// Create a one parameter constructor with parameter 'list' and type ArrayList.

   MultiThreadEx(ArrayList<Integer> list)

   {

     this.list = list;

   }

@Override

public void run()

{

  System.out.println("Run method");
```

```
  for(int i = 0 ; i < = 6; i++)

  {

// Adding elements in the list.

   list.add(i);

   try {

// Call sleep() method to delay some time.

     Threan ad.sleep(50);

   }

   catch(InterruptedException e)

   {

     e.printStackTrace();

   }

  }

}

public static void main(String[] args)

{

// Creates an ArrayList object with an initial capacity of 10.

   ArrayList<Ineger> list = new ArrayList<Integer>();



// Create multiple thread class objects m1, m2 and passes 'list' as a parameter.
```

```
   MultiThreadEx m1 = new MultiThreadEx(list);

   MultiThreadEx m2 = new MultiThreadEx(list);



// Call start() method to start thread using m1 and m2 reference variable.

    m1.start();

    m2.start();

   try {

// Call join() method to complete the task of adding elements in the ArrayList.

     m1.join();

     m2.join();

   }

   catch (InterruptedException e)

   {

      e.printStackTrace();

    }

Integer sizelist = list.size();

System.out.println("Size of list is " + sizelist);

for(Integer i : list)

{

 System.out.println("num - " + i);
```

```
}

 }

}
```

Output:

```
    Run method

    Run method

    Size of list is 12

    num - 0

    num - 0

    num - 1

    num - 2

    num - 2

    num - 3

    num - 3

    num - 4

    num - 4

    num - 5

    num - 5

    num - 6
```

Explanation:

Java Collections Notes

As you can observe in the preceding example program, ArrayList instance is sharing between two threads of class objects m1 and m2. Each thread is trying to add 6 elements in ArrayList. Since ArrayList is not synchronized.

Therefore, the expected output is unpredictable. When we run the above code, it shows the size of the ArrayList is 12 (not 14). Element 1 and 6 are inserted only one time in the list.

Let's see what happened in the above program.

➤ When thread-1 is created from the main thread, it inserts an element 0 in the list. After adding element 0 in the list, thread-1 goes to sleep position.

Since we are using join() method to complete the task of adding elements from 0 to 6 by m1, so when the thread-1 will go for the sleep, meanwhile, the main thread will start thread-2 for execution and will add an element 0 in the list.

➤ When the sleep time of thread-1 is over, thread-1 will again enter into the running state and add element 1 in the list. After adding, it will again go to sleep position.

➤ After sleeping time over, it again entered into the running state to insert element 2. After adding, thread-1 will again enter into sleep. Meanwhile, the main thread will again start thread-2 to complete its task and add element 2 which is unpredictable.

As you can observe in the program that the use of ArrayList in the multithreading environment does not provide thread safety. This is because ArrayList is not synchronized.

For more detail of Thread synchronization with diagram: Thread synchronization in Java

Note: If you run this program on your system, may be you will get different unpredictable output.

Java Collections Notes

Store User-defined Class Objects in ArrayList

In this tutorial, we will learn how to store user-defined (custom) class objects in Java ArrayList in an easy way and step by step.
In the previous ArrayList tutorial, we have learned that ArrayList class uses generic from Java 1.5 or later. Using generic, ArrayList class can be used to store any type of object.

In other words, we can store multiple types of objects in an ArrayList using generic feature.

For example, we could have an ArrayList of Book objects, an ArrayList of Employee objects, or an ArrayList of Strings. The specified type must be a class, not a primitive type.

If you want to store primitive data types in ArrayList, you will have to use one of java wrapper classes such as Integer, Double, or Character.

Declaration of ArrayList Objects

The syntax for declaring an ArrayList of objects is as follows:

```
ArrayList<ClassName> arrayListName;
```

Inside angle brackets, we declare the class types of objects that will be stored in ArrayList. Let's understand with the help of some examples.

For examples:

```
1. ArrayList<Employee> emp = new ArrayList<Employee>();



// Here, Employee is the name of class.
```

```
// ArrayList<Employee> indicates that only Employee objects can be stored in the array list.



   2. ArrayList<Book> b = new ArrayList<Book>();
```

How to store User-defined (Custom) Class Objects in Java ArrayList?

Let's take an example program where we will store user-defined (custom) class objects in ArrayList.

Suppose there are three students whose name, id, rollNo have to add to the custom ArrayList. First, we will create a class Student.

In this class, we will define a constructor and three instance variables name, id, and rollNo of data type String, int, and int respectively. Look at the below program source code.

Program source code 1:

```
package customArrayList;

public class Student

{

// Create instance variables name, id, and rollNo of data type String, int, and int respectively.

   String name;

   int id;

   int rollNo;
```

```
// Create three parameters constructor with parameters name, id, and rollNo.

   Student(String name, int id, int rollNo)

   {

     this.name = name;

     this.rollNo = rollNo;

     this.id = id;

   }

}
```

Now create another class addingData in which we will store three Student class objects in the ArrayList.

```
package customArrayList;

import java.util.ArrayList;

public class AddingData

{

// Create a ArrayList method of generic type 'Student'.

   ArrayList<Student> studentData()

   {

// Create three objects of the class Student and pass arguments to the constructor.

    Student s1 = new Student("Deep", 1234, 04);

    Student s2 = new Student("Shubh", 4321, 20 );
```

```
    Student s3 = new Student("Riddhi", 1212, 02);



// Create the object of ArrayList of generic type 'Student'.

    ArrayList<Student> studentlist = new ArrayList<Student>();



// Now add Student objects in the ArrayList using reference variable studentlist.

     studentlist.add(s1);

     studentlist.add(s2);

     studentlist.add(s3);



// Return object reference variable 'studentlist' of the array list to the method
'studentValue'.

     return studentlist;

   }

}
```

Now create one more class to retrieve students data from the above class's studentData() method and iterate over them to get the student detail.

```
package customArrayList;

import java.util.ArrayList;

public class RetriveStudents
```

```
{

public static void main(String[] args)

{

// Call AddingData class by creating object of that class.

   AddingData data = new AddingData();



// Call studentData() method using reference variable data.

   ArrayList<Student> listst = data.studentData();



// Now iterate and display all the Student data.

   for(Student st:listst)

   {

     System.out.println("Student's name: " +st.name);

     System.out.println("Student ID " +st.id);

     System.out.println("Roll number: " +st.rollNo);

   }

  }

}
```

Output:

```
      Student's name: Deep
```

```
Student ID 1234

Roll number: 4



Student's name: Shubh

Student ID 4321

Roll number: 20



Student's name: Riddhi

Student ID 1212

Roll number: 2
```

The same program can also be done without using any constructor. Look at the below source code and follow all steps.

Program source code 2:

```
package customArrayList2;

public class Student

{

// Declare instance variables name, phyMarks, mathsMarks, chemMarks, total, and per.

  String name;

  int phyMarks;

  int mathsMarks;

  int chemMarks;
```

```
  int total;

  float per;

 }

package customArrayList2;

import java.util.ArrayList;

public class Studentdata

{

// Declare an ArrayList method of generic type Student.

  ArrayList<Student> addData()

  {

// Create two objects s1 and s2 of the student class and initialize the value of variables
using reference variable s1 and s2.

   Student s1 = new Student();

    s1.name = "Shubh";

    s1.phyMarks = 95;

    s1.mathsMarks = 100;

    s1.chemMarks = 90;

    s1.total = 95 + 100 + 90;

    s1.per = ((s1.total)*100)/300;
```

```
  Student s2 = new Student();

   s2.name = "Deep";

   s2.phyMarks = 80;

   s2.mathsMarks = 85;

   s2.chemMarks = 90;

   s2.total = 80 + 85 + 90;

   s2.per = ((s2.total)*100)/300;



// Create an ArrayList object of generic type Student.

   ArrayList<Student> al = new ArrayList<Student>();



// Call add() method to store student class objects in the array list using reference variable
al.

    al.add(s1);

    al.add(s2);

    return al;

  }

}

package customArrayList2;

import java.util.ArrayList;
```

```
public class RetriveStudentData

{

// Declare an instance method.

   void fetchStudentData()

   {

// Create an object of the Studentdata class.

   Studentdata stdata = new Studentdata();



// Call addData() method using reference variable stdata.

    ArrayList<Student> listst = stdata.addData();



// Now iterate and display all the student data.

// enhance for loop - for each loop.

   for(Student student:listst)

   {

     System.out.println("Name: " +student.name);

     System.out.println("Physics Marks: " +student.phyMarks);

     System.out.println("Maths Marks: " +student.mathsMarks);



     System.out.println("Chemistry Marks: " +student.chemMarks);
```

```
    System.out.println("Total Marks: " +student.total);

    System.out.println("Percentage:" +student.per);

   }

 }

}
```

In this example program, we will create a client class to test our logic.

```
package customArrayList2;

public class TestStudentOperation

{

public static void main(String[] args)

{

  Studentdata st = new Studentdata();

   st.addData();



  RetriveStudentData rsd = new RetriveStudentData();

   rsd.fetchStudentData();

 }

}

Output:

      Name: Shubh
```

```
Physics Marks: 95

Maths Marks: 100

Chemistry Marks: 90

Total Marks: 285

Percentage: 95.0



Name: Deep

Physics Marks: 80

Maths Marks: 85

Chemistry Marks: 90

Total Marks: 255

Percentage: 85.0
```

Let's take another example program where we will create a class Employee and we will iterate and display all employee data from the Employee class. So let's see the following source code.

Program source code 3:

```
package customArrayList3;

import java.util.ArrayList;

public class Employee

{

  int eNo;
```

```
 String name, address;

 Employee(int eNo, String name, String address)

 {

  this.eNo = eNo;

  this.name = name;

  this.address = address;

 }

public Employee()

{



 }

// Display all employees data.

  void displayData(ArrayList<Employee> list)

  {

    System.out.println("Employee Detail");

    for(Employee emp: list )

    {

      System.out.println("Employee number: " +emp.eNo);

      System.out.println("Employee Name: " +emp.name);

      System.out.println("Employee Address: " +emp.address);
```

```
    }

   }

  }
package customArrayList3;

import java.util.ArrayList;

public class AddingEmployeeData

{

public static void main(String[] args)

{

  Employee emp1 = new Employee(102, "Shubh", "Nagpur" );

  Employee emp2 = new Employee(205, "Anjali", "Dhanbad");

  Employee emp3 = new Employee(333, "Shanjna", "Mumbai");



  ArrayList<Employee> list = new ArrayList<Employee>();

   list.add(emp1);

   list.add(emp2);

   list.add(emp3);

 Employee temp = new Employee();
```

```
// Call displayData method using temp reference variable object and pass list as a
parameter.

   temp.displayData(list);

  }

}
```

Output:

```
    Employee Detail

    Employee number: 102

    Employee Name: Shubh

    Employee Address: Nagpur



    Employee number: 205

    Employee Name: Anjali

    Employee Address: Dhanbad



    Employee number: 333

    Employee Name: Shanna

    Employee Address: Mumbai
```

LinkedList in Java | Methods, Example

Java Collections Notes

➲ **LinkedList in Java** is a linear data structure that uses a doubly linked list internally to store a group of elements.
A doubly linked list consists of a group of nodes that together represents a sequence in the list. It stores the group of elements in the sequence of nodes.

➲ Each node contains three fields: a data field that contains data stored in the node, left and right fields contain references or pointers that point to the previous and next nodes in the list.

A pointer indicates the addresses of the next node and the previous node. Elements in the linked list are called **nodes**.
➲ Since the previous field of the first node and the next field of the last node do not point to anything, we must set it with the null value.

➲ LinkedList in Java is a very convenient way to store elements (data). When we store a new element in the linked list, a new node is automatically created.

Its size will grow with the addition of each and every element. Therefore, its initial capacity is zero. When an element is removed, it will automatically shrink.

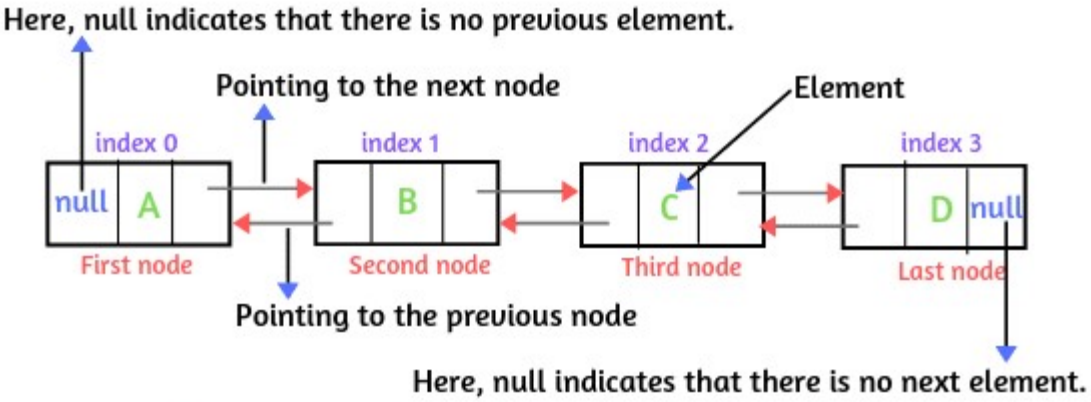
Adding elements into the LinkedList and removing elements from the LinkedList are done quickly and take the same amount of time (i.e. constant time).

So, it is especially useful in situations where elements are inserted or removed from the middle of the list.

➲ In the Linked List, the elements are not stored in the consecutive memory location. An element (often called node) can be located anywhere in the free space of the memory by connecting each other using the left and right sides of the node portion.

An array representation of linear doubly LinkedList in Java is shown in the below figure.

Representation of Java LinkedList Node



A array representation of linear Doubly LinkedList in Java

Thus, it avoids the rearrangement of elements required but requires that each element is connected to the next and previous by a link.

Therefore, a LinkedList is often a better choice if elements are added or removed from intermediate locations within the list.

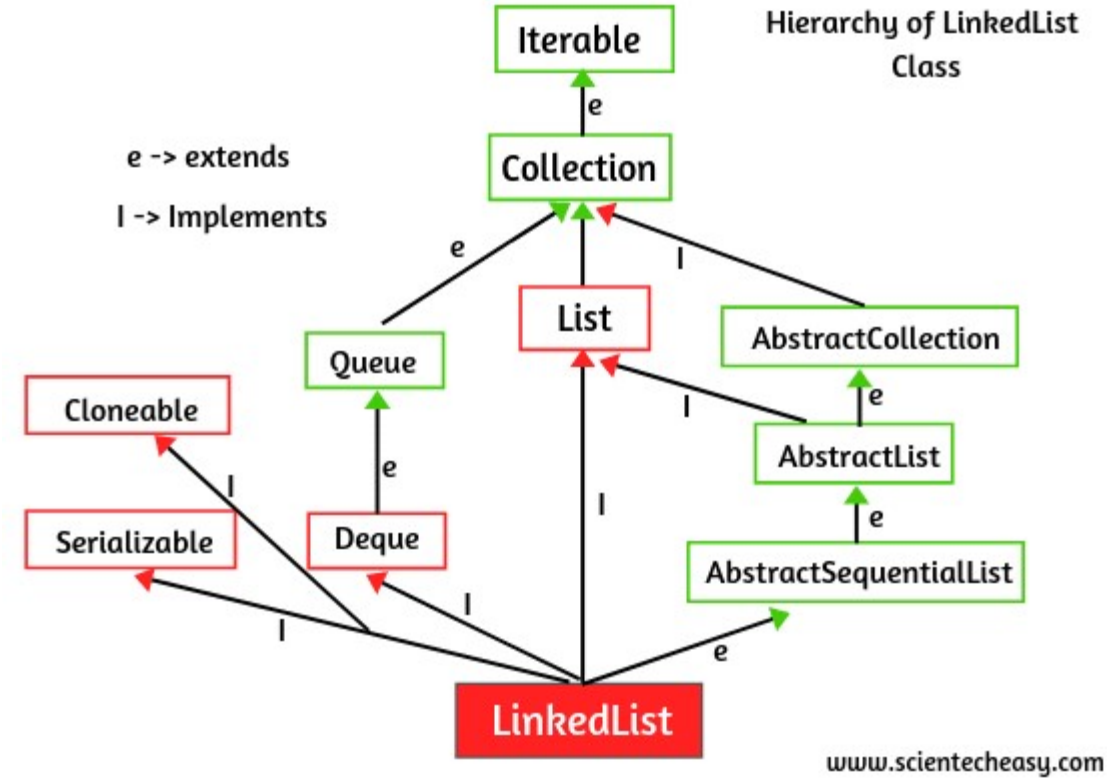
The right side of the node contains the address of the next element and the left side of the node contains the address of the previous element in the list.

Hierarchy Diagram of LinkedList class in Java

Java LinkedList class implements List, Deque, and Queue interfaces. It extends AbstractSequentialList. It also implements marker interfaces such as serializable and cloneable but does not implement random access interface.

Java Collections Notes

The hierarchy diagram of the LinkedList class in Java can be shown in the below figure.



Java LinkedList class declaration

LinkedList is a generic class, just like ArrayList class that can be declared as:

```
class LinkedList<E>
```

Here, E specifies the type of elements (objects) in angle brackets that the linked list will hold. For example, LinkedList<String> or LinkedList<Employee>.

LinkedList class was introduced in Java 1.2 version and it is placed in java.util package.

Java Collections Notes

Java LinkedList Constructors

Like ArrayList class, Java LinkedList class consists of two constructors. They are listed in the below table.

SN	Constructor	Description
1.	LinkedList()	It is used to create an empty LinkedList object.
2.	LinkedList(Collection c)	It is used to construct a list containing the elements of the given collection.

We can create an empty linked list object for storing String type elements (objects) as:

```
LinkedList<String> llist = new LinkedList<String>(); // An empty list.
```

Features of LinkedList class

The main features of the Java LinkedList class are as follows:

1. The underlying data structure of LinkedList is a doubly LinkedList data structure. It is another concrete implementation of the List interface like an array list.

2. Java LinkedList class allows storing duplicate elements.

3. Null elements can be added to the linked list.

4. Heterogeneous elements are allowed in the linked list.

5. Java LinkedList is not synchronized. So, multiple threads can access the same LinkedList object at the same time. Therefore, It is not thread-safe. Since LinkedList is not synchronized. Hence, its operation is faster.

6. Insertion and removal of elements in the LinkedList are fast because, in the linked list, there is no shifting of elements after each adding and removal. The only reference for next and previous elements changed.

7. LinkedList is the best choice if your frequent operation is insertion or deletion in the middle.

8. Retrieval (getting) of elements is very slow in LinkedList because it traverses from the beginning or ending to reach the element.

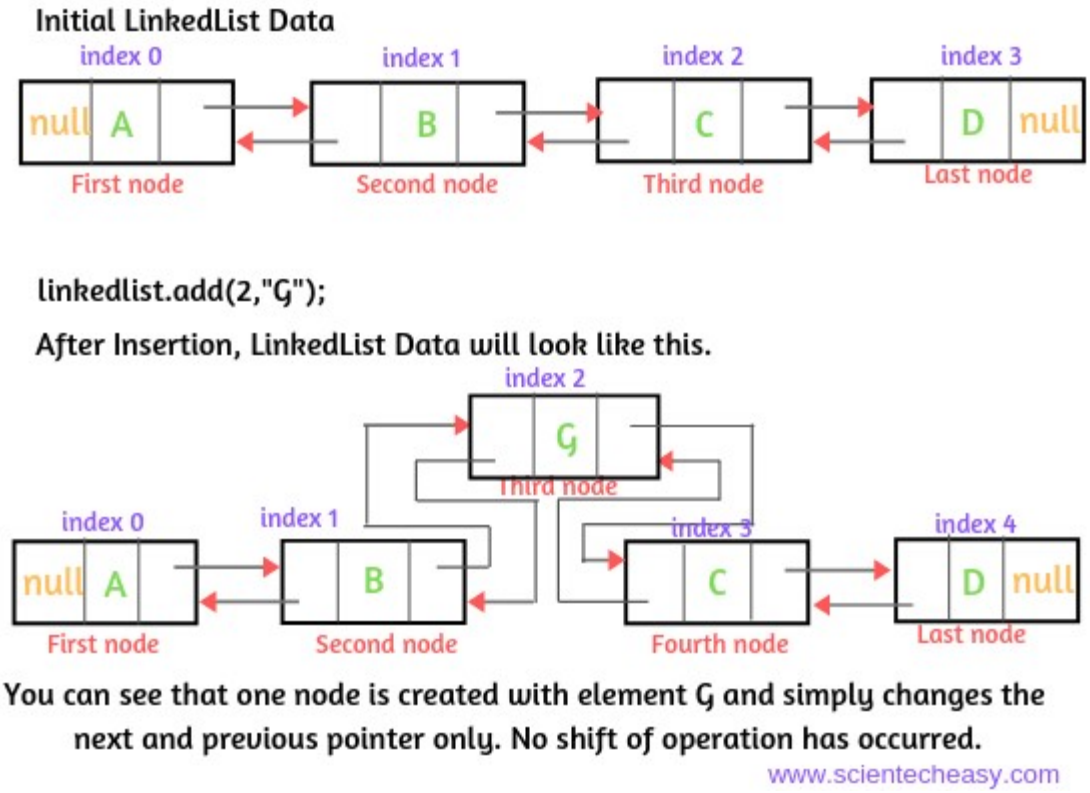
9. The LinkedList can be used as a "stack". It has pop() and push() methods which make it function as a stack.
10. Java LinkedList does not implement random access interface. So, the element cannot be accessed (getting) randomly. To access the given element, we have to traverse from the beginning or ending to reach elements in the LinkedList.

11. We can iterate linked list elements by using ListIterator.

How does Insertion work in Java LinkedList?

In the Java LinkedList, we can perform insertion (addition) operations without affecting any of data items already stored in the list. Let's take an example to understand this concept.

1. Let us assume that our initial LinkedList has the following data as shown in the below figure.



2. Now we will perform insertion operation on this linked list. We will add an element G at index position 2 using add() method. The syntax for adding element G is as follow:

```
linkedlist.add(2,"G");
```

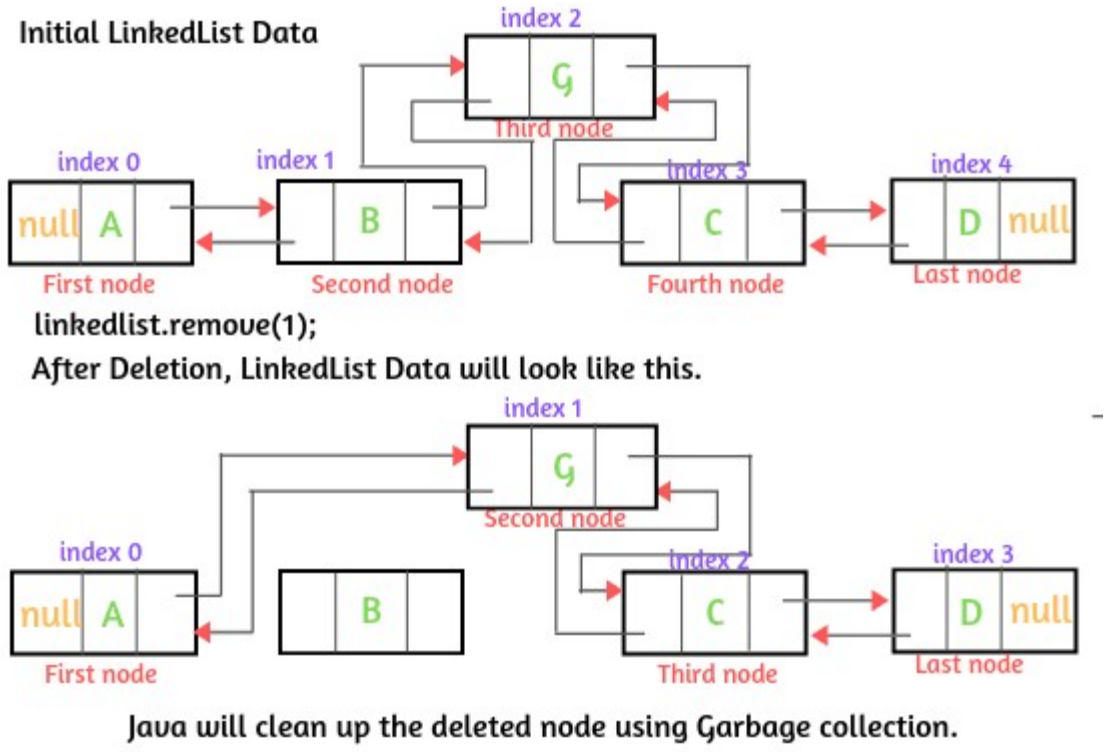
When the insertion operation takes place in the linked list, internally, LinkedList will create one node with G element at anywhere available space in the memory and changes the next and previous pointer only without shifting of any element in the list. Look at the updated LinkedList in the above figure.

How does Deletion work in Java LinkedList?

In the previous section, we have already seen how linked list performs insertion operations internally. Now we will discuss how Java linked list performs deletion operation internally.

Java Collections Notes

1. Let us assume that our initial LinkedList has the following data.



2. We will perform deletion operations on this LinkedList. Suppose we want to remove or delete element B from index position 1. The syntax to delete element B is as follows:

```
linkedlist.remove(1);
```

When deletion operation takes place, the node with element B is deleted with changing the next and previous pointer. The deleted node becomes unused memory.

Therefore, Java will clean up the unused memory space using the garbage collection.

LinkedList Methods in Java

Java Collections Notes

In addition to implementing the List interface, Java LinkedList class also provides methods for retrieving, inserting, and removing elements from both ends of the list. LinkedList methods are listed in the below table:

SN	Method	Description
1.	boolean add(Object o)	It is used to add the specified element to the end of a list.
2.	void add(int index, Object o)	It is used to insert the specified element at the specified position in a list.
3.	boolean addAll(Collection c)	It is used to add all of the elements in the specified collection to the end of this list
4.	boolean addAll(int index, Collection c)	It is used to add all the elements of the specified collection at specified index position in the list.
5.	void clear()	It is used to remove or delete all the elements from a list.
6.	boolean contains(Object o)	It returns true if a list contains a specified element.
7.	int size()	It is used to get the number of elements in a list.
8.	Object[] toArray()	It returns an array containing all the elements in a list in proper sequence
9.	Object clone()	This method is used to return a shallow copy of an ArrayList.
10.	boolean remove(Object o)	This method is used to remove the first occurrence of the specified element in the linked list.
11.	Element remove(int index)	This method is used to remove the element at the specified position in the linked list.
12.	Element element()	This method is used to retrieve the first element of the linked list.
13.	E get(int index)	It is used to get the element at the specified position in a list.

14.	Element set(int index, E element)	It is used to replace the element at the specified position in a list with the specified element.
15.	int indexOf(Object o)	It is used to get the index of the first occurrence of the specified element in the list. It returns -1 if the list does not contain any element.
16.	int lastIndexOf(Object o)	It is used to get the index of the last occurrence of the specified element in a list. It returns -1 if the list does not contain any element.
17.	Iterator iterator()	This method returns an iterator over the elements in a proper sequence in the linked list.
18.	ListIterator listIterator(int index)	This method returns a list-iterator of the elements in a proper sequence, starting at the specified position in the list.

LinkedList Deque Methods

Java LinkedList class has also various specific methods that are inherited from Deque interface. They are listed in the below table:

SN	Methods	Description
1.	void addFirst(Object o)	It is used to add the specified element in the first position of the LinkedList.
2.	void addLast(Object o)	It is used to add the specified element to the end of the LinkedList.
3.	Object getFirst()	This method is used to return the first element from the list.
4.	Object getLast()	It is used to get the last element from the list.
5.	Object removeFirst()	It removes the first element from the linked list and returns it.
6.	Object removeLast()	It removes the last element from the linked list and returns it.

7.	boolean offerFirst(Object o)	It is used to insert the specified element at the front of a list.
8.	boolean offerLast(Object o)	It is used to insert the specified element at the end of a list.
9.	Object peek()	It retrieves the first element from the linked list
10.	Object peekFirst()	It is used to retrieve the first element from the linked list. It will return null if the list is empty.
11.	Object peekLast()	It is used to retrieve the last element from the linked list. It will return null if the list is empty.
12.	Object poll()	It retrieves and removes the first element from the linked list.
13.	Object pollFirst()	It retrieves and removes the first element from the linked list. It returns null if a list is empty.
14.	Object pollLast()	It retrieves and removes the last element from the linked list. It returns null if a list is empty.
15.	Object pop()	This method is used to pop an element from the stack represented by a list.
16.	void push(Object o)	This method is used to push an element onto the stack represented by a list.

What is best case scenario to use LinkedList in Java application?

Java LinkedList is the best choice to use when your frequent operation is adding or removing elements in the middle of the list because the adding and removing of elements in the linked list is faster as compared to ArrayList.

Let's take a realtime scenario to understand this concept. Suppose there are 100 elements in the ArrayList. If we remove the 50th element from the ArrayList, 51st element will go to 50th position, 52nd element to 51st position, and similarly for other elements that will consume a lot of time for shifting. Due to which the manipulation will be slow in ArrayList.

But in the case of linked list, if we remove 50th element from the linked list, no shifting of elements will take place after removal. Only the reference of the next and previous node will change.

Moreover, LinkedList can be used when we need a stack (LIFO) or queue (FIFO) data structure by allowing duplicates.

What is worst case scenario to use LinkedList?

Java LinkedList is the worst choice to use when your frequent operation is retrieval (getting) of elements from the linked list because retrieval of elements is very slow in the LinkedList as compared to ArrayList.

Since LinkedList does not implement Random Access Interface. Therefore, an element cannot be accessed (getting) randomly. We will have to traverse from the beginning or ending to reach elements in the linked list.

Let us consider a realtime scenario to understand this concept.

Assume that there are ten elements in the list. Suppose that LinkedList accesses the first element in one second from the list and retrieves the element.

Since the address of second element is available in the first node. So, it will take two seconds time to access and get the second element.

Similarly, if we want to get 9th element from the list then LinkedList will take 9 sec time to get element. Why?

This is because the address of 9th element is available in 8th node. The address of 8th node is available in 7th node and so on.

But if there are one crore elements in the list and we want to get 50th lakh element from the list, may be linked list will take one year time to access and getting 50th lakh element.

Therefore, LinkedList is the worst choice for the retrieval or searching of elements from the linked list.

In this case, ArrayList is the best choice to use for getting elements from the list because ArrayList implements random access interface.

So, we can get an element from the array list very fast from any arbitrary position.

Hope that this tutorial has covered almost all the important topics related to **LinkedList in Java** and its various methods with some example.

Linked List Program in Java for Best Practice

In this tutorial, we will see different kinds of linked list example program in Java with the best explanation in a simple way.

If you have any doubt related to the basics of linked list, recommends that you once visit LinkedList tutorial, clear whatever doubts, and then come to practice example programs.

How to add Element to LinkedList in Java?

Let's take an example program where we will add elements in different ways using add(), add(int index, Object o), addFirst(Object o),

addLast(Object o), addAll() methods of list interface and deque interface. Follow all steps in the coding.

Program source code 1:

```
package linkedListProgram;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Vector;

public class AddingExample

{

public static void main(String[] args)

{

// Create a LinkedList object.

   LinkedList list = new LinkedList(); // empty linked list.



// Call add() method for adding Heterogeneous elements using reference variable list.

   list.add("One");

   list.add(2);

   list.add(null); // null elements are allowed in the linked list.

   list.add("Four");

   System.out.println("Initial LinkedList order: " +list);
```

```
// Adding an element specified in the linked list.

   list.add(3, "Three");



// Adding an element at the first position of list using addFirst() method of Deque
interface.

   System.out.println("LinkedList Elements after adding the first element");

   list.addFirst("Zero");

   System.out.println(list);



// Adding an element at the end of the list using addLast() method of Deque interface.

   System.out.println("LinkedList Elements after adding the last element");

   list.addLast("Five");

   System.out.println(list);



// Adding all elements from existing ArrayList collection to the end of the LinkedList.

// Create an ArrayList object.

   ArrayList al = new ArrayList();

    al.add("Six");

    al.add(7);

    al.add("Eight");
```

```
// Call addAll() method to add all elements to the end of the linked list.
   list.addAll(al);
   System.out.println("LinkedList Elements after adding all elements from ArrayList");
   System.out.println(list);

// Adding all elements from an existing Vector collection at the specified position of LinkedList.
   Vector v = new Vector();
    v.add(7.5);
    v.add(7.8);
   list.addAll(9, v);
   System.out.println("Linkedlist elements after adding all elements from vector");
   System.out.println(list);
  }
}
```

Output:

```
      Initial LinkedList order:
      [One, 2, null, Four]
      LinkedList Elements after adding the first element
```

```
[Zero, One, 2, null, Three, Four]

LinkedList Elements after adding the last element

[Zero, One, 2, null, Three, Four, Five]

LinkedList Elements after adding all elements from ArrayList

[Zero, One, 2, null, Three, Four, Five, Six, 7, Eight]

Linked list Elements after adding all elements from the vector

[Zero, One, 2, null, Three, Four, Five, Six, 7, 7.5, 7.8, Eight]
```

How to remove Element from LinkedList in Java?

Let's take an example program and see how to remove elements from the list in different ways using methods such as remove(), removeFirst(), removeLast().

Program source code 2:

```
package linkedListProgram;

import java.util.ArrayList;

import java.util.LinkedList;

public class RemovingElementExample

{

public static void main(String[] args)

{

// Create a generic LinkedList object of type String.
```

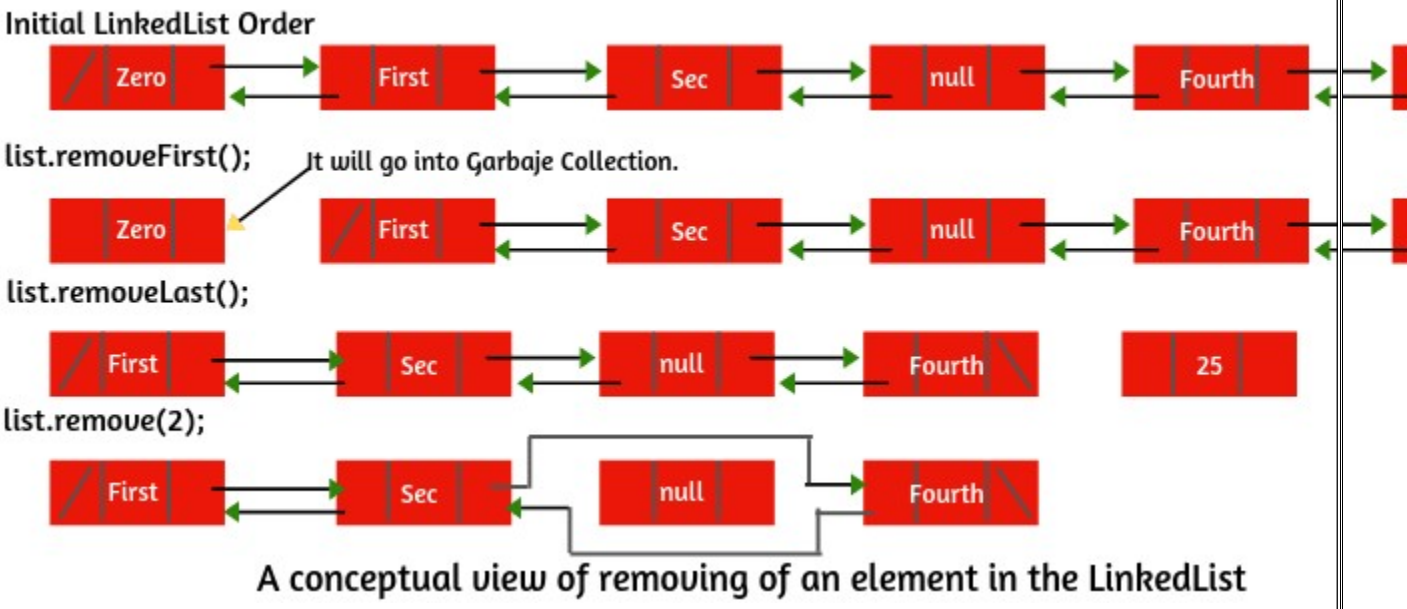
```
   LinkedList<String> list = new LinkedList<String>();

   int size = list.size();

   System.out.println("Size of Linkedlist: " +size);


// Adding elements of String type.

   list.add("Zero");

   list.add("First");

   list.add("Second");

   list.add(null); // null elements are allowed in the linked list.

   list.add("Fourth");

   list.add("25");

  System.out.println("Initial LinkedList order: " +list);


// Removing the first element from list using removeFirst() method.

   list.removeFirst();

   System.out.println("LinkedList Elements after removing the first element");

   System.out.println(list);


// Removing the last element from the list using removeLast() method.

   System.out.println("LinkedList Elements after removing the last element");
```

```
   list.removeLast();

   System.out.println(list);



// Removing element at the specified position from the list.

   list.remove(2);

   System.out.println("LinkedList Elements after removing the element at index position 2
");

   System.out.println(list);



// Adding all elements from existing an ArrayList collection to the end of LinkedList.

// Creating a generic ArrayList object of String type.

   ArrayList<String> al = new ArrayList<String>();

    al.add("Third");

    al.add("Fourth");



  list.addAll(2, al);

  list.removeLastOccurrence("Fourth");



  System.out.println("LinkedList Elements after removing last occurrence");

  System.out.println(list);
```

```
}

}
```

A conceptual view of removing an element in LinkedList is shown in the below figure.

a href="https://www.scientecheasy.com/2020/09/java-linkedlist-program.html/">



A conceptual view of removing of an element in the LinkedList

```
Output:

    Size of LinkedList: 0

    Initial LinkedList order: [Zero, First, Second, null, Fourth, 25]

    LinkedList Elements after removing the first element

    [First, Second, null, Fourth, 25]

    LinkedList Elements after removing the last element

    [First, Second, null, Fourth]

    LinkedList Elements after removing the element at index position 2
```

```
[First, Second, Fourth]

LinkedList Elements after removing last occurrence

[First, Second, Third, Fourth]
```

How to get Element from LinkedList in Java?

Let's create a program where we will get an element from the list in different ways using get(), getFirst(), and getLast() method.

Program source code 3:

```
package linkedListProgram;

import java.util.LinkedList;

public class GetElementExample

{

public static void main(String[] args)

{

// Create a LinkedList object.

   LinkedList<Integer> list = new LinkedList<Integer>();

// Adding even numbers from 0 to 20 as elements in the list.

   for(int i = 0 ; i < = 20; i++)

   {

    if(i % 2 == 0) // It will check even number.
```

```
        list.add(i);

    }

// Call getFirst() method to get first even number.

    Object firstEvenNumber = list.getFirst(); // Return type of getFirst() methods is an
Object.

    System.out.println("First even number: " +firstEvenNumber);



  Object lastEvenNumber = list.getLast();

  System.out.println("Last even number: " +lastEvenNumber);

  Object getElement = list.get(5);

  System.out.println("Even number at index 5: " +getElement);

 }

}
```

Output:

```
        First even number: 0

        Last even number: 20

        Even number at index 5: 10
```

How to retrieve and remove Element in the Java LinkedList?

Java Collections Notes

We can retrieve and remove an element from LinkedList using peekFirst(), peekLast(), pollFirst(), and pollLast() methods.

1. peekFirst(): This method retrieves the first element from the list but does not remove the element from the list.
2. peekLast(): This method retrieves the last element from the list but does not remove it.
3. pollFirst(): This method retrieves and removes the first element from the list.
4. pollLast(): This method retrieves and removes the last element from the list.

Program source code 4:

```
package linkedListExample;

import java.util.LinkedList;

public class PeakPollExample

{

public static void main(String[] args)

{

// Create a LinkedList object.

   LinkedList<String> list = new LinkedList<String>();



// Adding elements to the list.

   list.add("INDIA");

   list.add("USA");

   list.add("JAPAN");
```

```
   list.add("UK");

   list.add("CANADA");

   System.out.println("Initial LinkedList order");

   System.out.println(list);



// Call peek() method to retrieve the first element from list.

   Object firstElement = list.peekFirst(); // Return type of this method is an Object.

   System.out.println("Retrieve the first element: " +firstElement);

   Object lastElement = list.peekLast();

   System.out.println("Retrieve the last element: " +lastElement);



// Call pollLast() to retrieve and remove the last element from the list.

   Object element1 = list.pollLast();

   System.out.println("Retrieve and remove the last element: " +element1);

   System.out.println("LinkedList Element after using pollLast() method");

   System.out.println(list);

  }

}
```

Output:

```
      Initial LinkedList order
```

```
[INDIA, USA, JAPAN, UK, CANADA]

Retrieve the first element: INDIA

Retrieve the last element: CANADA

Retrieve and remove the last element: CANADA

LinkedList Element after using pollLast() method

[INDIA, USA, JAPAN, UK]
```

How to Pop and Push Element from and onto stack in Java LinkedList?

Pop() method: This method is equivalent to removeFirst() method that removes and returns the first element from the list. In other words, It pops an element from the stack represented by the list.
Push() method: This method is equivalent to addFirst() method that adds an element at the first position of the list. In other words, It pushes an element onto the stack represented by the list.
Let's make a program where we will pop and push an element from and onto stack in LinkedList.

Program source code 5:

```
package linkedListProgram;

import java.util.LinkedList;

public class PeakPollExample

{

public static void main(String[] args)
```

```
{

 LinkedList<Character> list = new LinkedList<Character>();



// Adding elements in the list.

   list.add('A');

   list.add('B');

   list.add('C');

   list.add('D');

   list.add('E');



  System.out.println("Initial LinkedList order");

  System.out.println(list);

  Object element = list.pop(); // Return type of this method is an Object.



 System.out.println("Pops Element: " +element);

  list.push('B');

 System.out.println("LinkedList Element after pushing");

 System.out.println(list);

 }

}
```

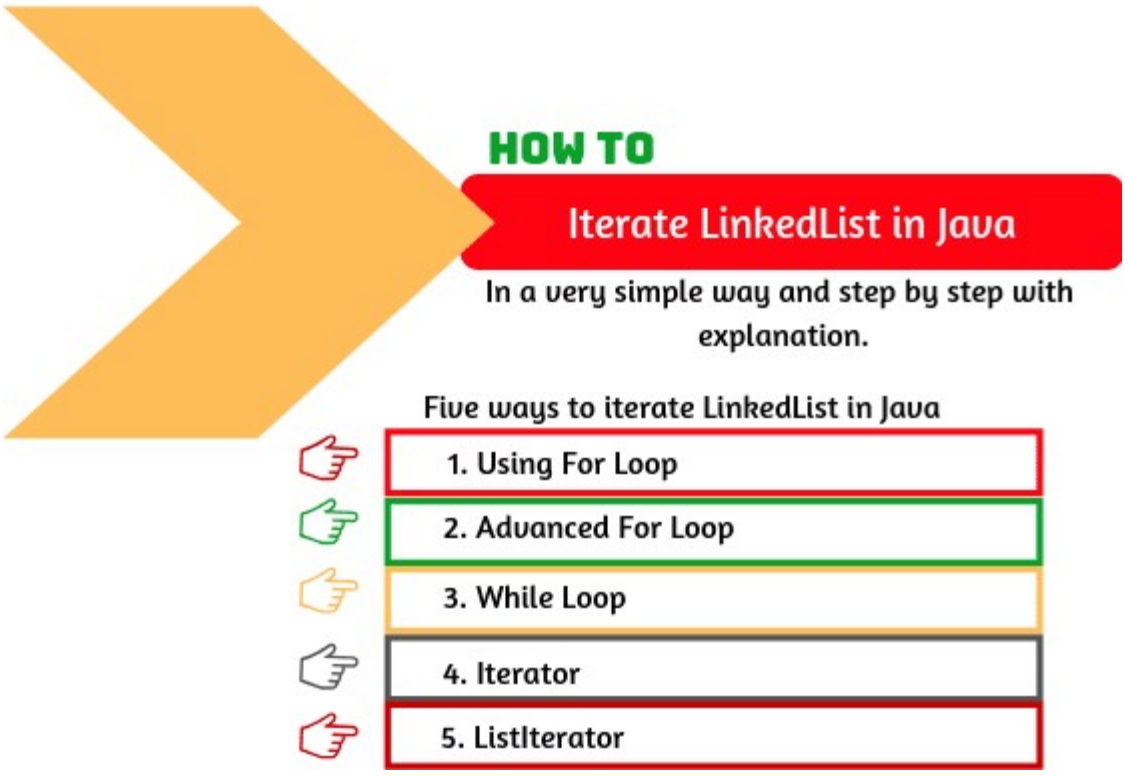
```
Output:

    Initial LinkedList order

    [A, B, C, D, E]

    Pops Element: A

    LinkedList Element after pushing

    [B, B, C, D, E]
```

How to Iterate LinkedList in Java

In the last tutorial, we have discussed Java LinkedList and its various methods with various example programs. In this tutorial, we will learn **how to iterate LinkedList in Java**.
Before going on this topic, I will recommend that you first clear all the basics of LinkedList in Java then come on this topic.
There are five ways in which LinkedList can be iterated in Java. They are as follows:

1. For loop
2. Advanced For loop
3. While loop
4. Iterator
5. ListIterator

Java Collections Notes



Iterating LinkedList in Java using For loop, Advanced For loop, & While loop

---

Let's create a program where we will iterate LinkedList using for loop, advanced for loop, and while loop.

Program source code 1:

```
package iterateLinkedList;

import java.util.LinkedList;

public class LinkedListEx1 {

public static void main(String[] args)

{
```

```
// Create a generic LinkedList object of String type.
   LinkedList<String> list = new LinkedList<String>(); // An empty list.


// Adding elements in the list.
    list.add("Red");
    list.add("Yellow");
    list.add("Green");
    list.add("White");


// Iterating using for loop.
  System.out.println("**For loop**");
  for(int i = 0; i < list.size(); i++)
  {
    Object element = list.get(i); // Return type of get() method is an Object.
    System.out.println(element);
}
// Iterating using Advanced for loop.
   System.out.println("**Advanced For loop**");
   for(String str: list)
   {
```

```
    System.out.println(str);
   }
// Iterating using while loop.
   System.out.println("**While Loop**");
   int num = 0;
  while (list.size() > num)
  {
    System.out.println(list.get(num));
     num++;
   }
 }
}
Output:
      **For loop**
      Red Yellow Green White
      **Advanced For loop**
      Red Yellow Green White
      **While Loop**
      Red Yellow Green White
```

How to iterate Java LinkedList using Iterator?

Let's take an example program where we will iterate elements of the linked list using Iterator. Using Iterator, we will iterate the list in the forward direction only. The steps are given in the following source code.

Program source code 2:

```
package iterateLinkedList;

import java.util.Iterator;

import java.util.LinkedList;

public class LinkedListEx2 {

public static void main(String[] args)

{

// Create a generic LinkedList object of Character type.

   LinkedList<Character> list = new LinkedList<Character>();



// Adding elements in the list.

    list.add('A');

    list.add('B');

    list.add('C');

    list.add('D');

    list.add('E');
```

```
// Iterating using Iterator.

   System.out.println("**Using Iterator**");

   Iterator<Character> itr = list.iterator();

  while(itr.hasNext())

  {

   Object obj = itr.next();

   System.out.println(obj);

   }

  }

}

Output:

      **Using Iterator**

      A B C D E
```

How to traverse LinkedList in Java using ListIterator?

Let's take an example program where we will traverse or iterate elements of the LinkedList using ListIterator. We can iterate elements of the list in both forward as well backward directions. Look at the following source code to understand better.

Program source code 3:

```
package iterateLinkedList;

import java.util.LinkedList;
```

```
import java.util.ListIterator;

public class LinkedListEx3 {

public static void main(String[] args)

{

// Create a generic LinkedList object of type Integer.

   LinkedList<Integer> list = new LinkedList<Integer>();



// Adding elements in the list.

    list.add(10);

    list.add(20);

    list.add(30);

    list.add(40);

    list.add(50);



System.out.println("LinkedList original order");

System.out.println(list);



ListIterator<Integer> litr = list.listIterator();

System.out.println("Interating in forward direction");

 while(litr.hasNext())
```

```
{

 Object obj = litr.next();

 System.out.println(obj);

}

System.out.println("Iterating in backwrd direction");

while(litr.hasPrevious())

{

 Object obj1 = litr.previous();

 System.out.println(obj1);

  list.add(60); // It will throw Concurrent Modification Exception because we cannot add or remove element in the LinkedList during iteration.

 }

System.out.println(list);

}

}
```

Output:

```
    LinkedList original order

    [10, 20, 30, 40, 50]

    Interating in forward direction

    10 20 30 40 50
```

```
Iterating in backwrd direction

50

java.util.ConcurrentModificationException at
java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:966) at

java.util.LinkedList$ListItr.previous(LinkedList.java:903) at
iterateLinkedList.LinkedListEx3.main(LinkedListEx3.java:31)
```

Set in Java | Set Methods, Example

A **set in Java** is an unordered collection of unique elements or objects. In other words, a set is a collection of elements that are not stored in a particular order.
Elements are simply added to the set without any control over where they go. For example, in many applications, we do not need to care about the order of elements in a collection.

Consider the below figure where a set of books by topics is not arranged. A collection of elements is not stored in a particular order into a set.

Such a collection of elements without order is called set in Java.

Java Collections Notes

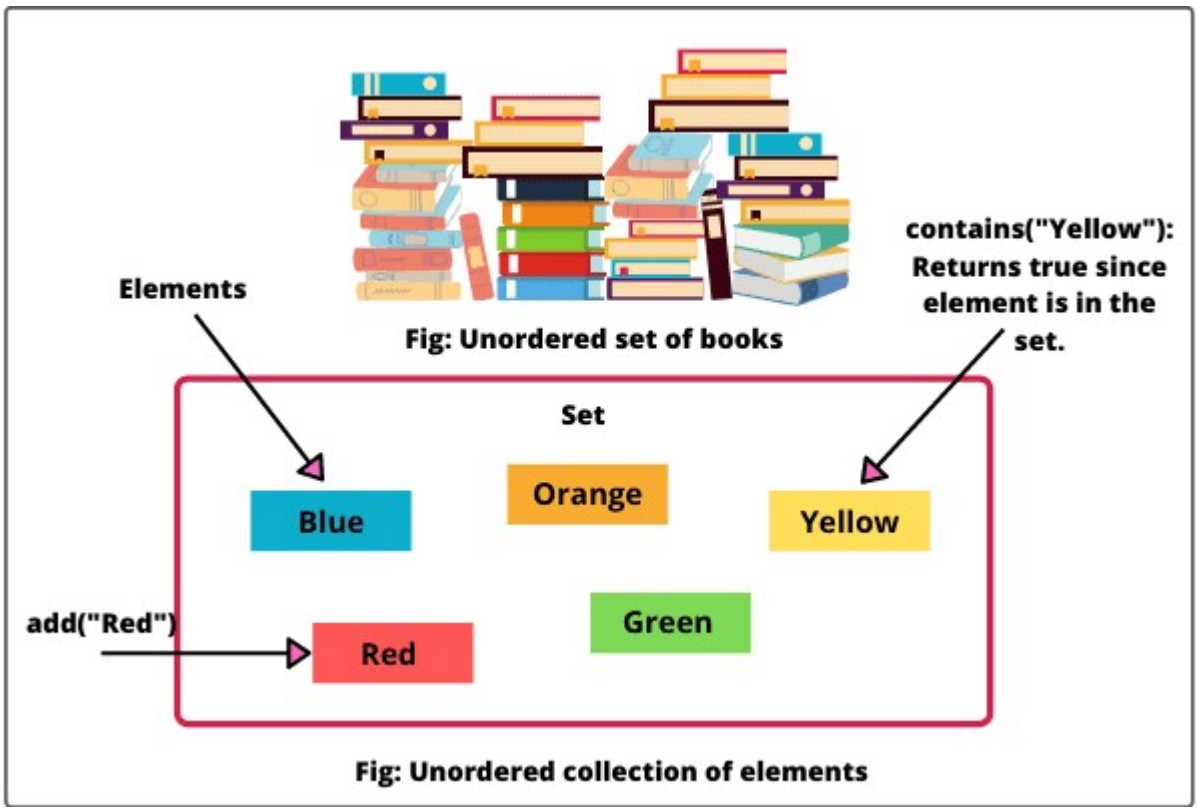


Fig: Unordered set of books

Fig: Unordered collection of elements

We can add elements in a set and iterate over all elements in a set. It will grow dynamically when elements are stored in it. We can also check whether a particular element is present in the set.
A set will not allow duplicate elements. That means an element can exist only once in the set. If we try to add the same element that is already present in the set, then it is not stored in the set. That's why each element in a set is unique.

Set in Java can be used to remove duplicate elements from the collection.

Set holds a single reference to an object. It does not provide two references to the same object, two references to null, or references to two objects a and b such that a.equals(b).

Set Interface in Java

Java Collections Notes

Set is an interface that was introduced in Java 1.2 version. It is a generic interface that is declared as:

```
interface Set<E>
```

Here, E defines the type of elements that the set will hold.

Java Set interface does not provide any additional methods of its own. It uses methods defined by the collection interface but places additional restrictions on those methods.

For example, when set uses add() or addAll() methods defined by the collection interface, it does not add an element to the set if the set already contains that element.

Java Set interface does not provide any get() method like List to retrieve elements. Therefore, the only way to take out elements from the set is to do using Iterator() method. But this method does not return elements from the set in any particular order.

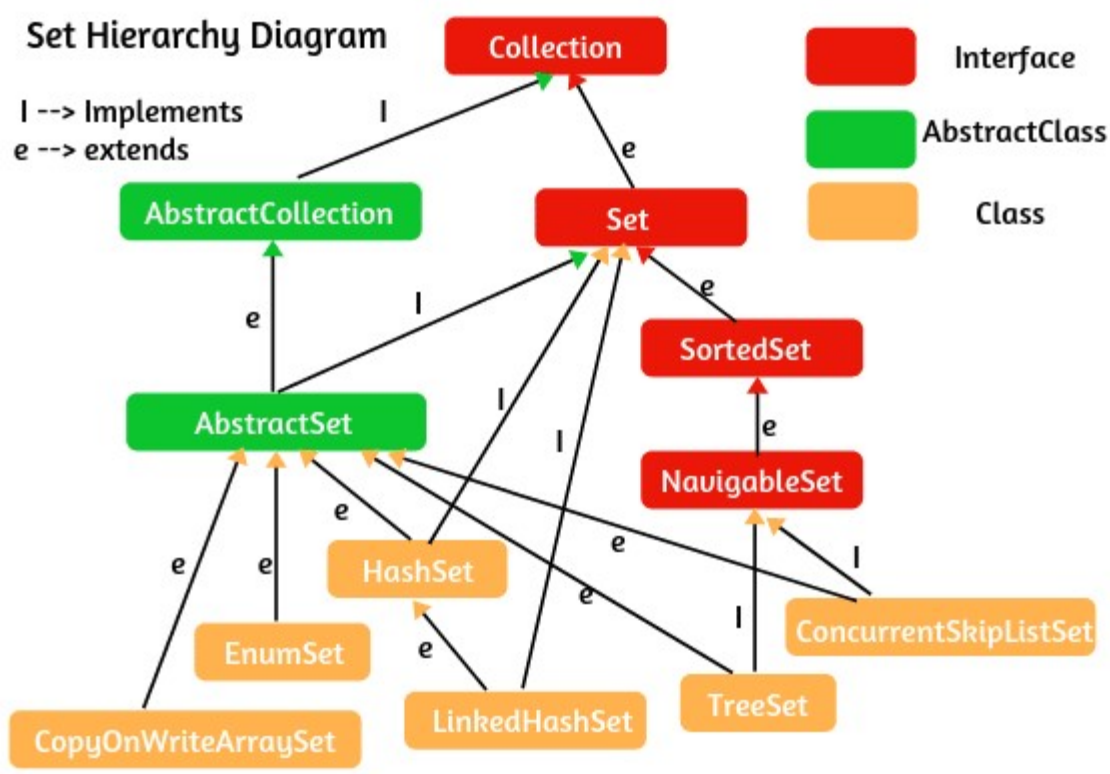
Using the Iterator, we can traverse only in the forward direction from the first to last element. We cannot traverse over elements in the backward direction using iterator method.

Hierarchy of Set Interface in Java

Java Set interface extends java.util.collection interface. The java.util.SortedSet interface extends the Set interface to provide the sorted set of elements.

Three classes such as HashSet, LinkedHashSet, and TreeSet implement set interface.

ConcurrentSkipListSet and EnumSet classes also implements set interface. The hierarchy diagram of the Set interface in Java is shown in below figure.



Features of Java Set

---

1. Set is an unordered collection of elements. That means the order is not maintained while storing elements. While retrieving we may not get the same order as that we put elements.

2. It is used to store a collection of elements without duplicate. That means it contains only unique elements.

3. Java Set uses map based structure for its implementation.

4. It can be iterated by using Iterator but cannot be iterated by using ListIterator.

5. Most of the set implementations allow adding only one null element. Tree set does not allow to add null element.

6. Set is not an indexed-based structure like a list in Java. Therefore, we cannot access elements by their index position.
8. It does not provide any get method like a list.

Java Set Implementation

Java collections framework provides three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.
HashSet is a concrete class that implements set interface. It uses a hash table mechanism to store its elements. It is the best-performing implementation.

TreeSet is a concrete class that implements SortedSet interface (a subinterface of set). It uses a binary search tree mechanism to store its elements. It orders its elements based on their values. It is considerably slower than HashSet.

LinkedHashSet is a concrete class that extends HashSet class. It stores its elements using hash table mechanism with a linked list implementation.

It orders its elements based on the order in which they were inserted into the set. That is, elements in the HashSet are not ordered but elements in the LinkedHashSet can be retrieved in the order in which they were inserted into the set.

This is the brief idea of set implementation classes. We will learn more detail about HashSet, TreeSet, and LinkedHashSet classes in further tutorials.

Set Methods in Java

Java Collections Notes

A set interface has the following various useful methods such as add, remove, clear, size, etc to enhance the usage of a set interface in the collections framework. They are listed in the table:

Method	Description
boolean add(Object o)	It is used to add the specified element in this set.
boolean addAll(Collection c)	This method adds all the elements in the given collection.
int size()	It is used to get the number of elements in the set.
boolean isEmpty()	This method checks that the set is empty or not.
void clear()	It is used to remove all the elements from the set.
boolean contains(Object o)	This method returns true if this set contains the given element.
boolean containsAll(Collection c)	This method returns true if this set contains all the elements of the given collection.
boolean remove(Object o)	It is used to remove the specified element from this set.
boolean removeAll(Collection c)	It removes all the elements in the given collection from the set.
Iterator iterate()	It returns an Iterator over the elements in this set.
boolean equals(Object o)	It is used to compare the given element for equility in this set.
int hashCode()	It is used to get the hashCode value for this set.

Ways to create a Generic Set object in Java

Java Collections Notes

We can create a generic Set object by using one of its three concrete classes: HashSet, LinkedHashSet, and TreeSet. The syntax to create set objects is as follows:

Syntax:

```
Set<T> set = new HashSet<T>(); where T is type of generic.

Set<T> set = new LinkedHashSet<T>();

Set<T> set = new TreeSet<T>();



For example:

   Set<Integer> set = new HashSet<Integer>(); // Creates an empty set of Integer objects.

   Set<String> set2 = new LinkedHashSet<String>(); // Creates an empty set of String objects.
```

Set Example Programs based on basic Operations

1. Adding Elements to Set: The add() method returns true if set does not contain the specified element. If set already contains the specified element then it will return false. Let's take an example program based on it.

Program source code 1:

```
package setProgram;

import java.util.HashSet;

import java.util.LinkedHashSet;
```

```
import java.util.Set;

public class SetExample1

{

public static void main(String[] args)

{

// Create a generic set object of type String.

   Set<String> s = new HashSet<String>();

   int size = s.size();

  System.out.println("Size before adding elements: " +size);



// Adding elements to set.

   s.add("Orange"); // s.size()is 1.

   s.add("Red"); // s.size() is 2.

   s.add("Blue"); // s.size() is 3.

   s.add("Yellow"); // s.size() is 4.

   s.add("Green"); // Now s.size() is 5.



// Add duplicate elements in the set. These elements will be ignored by set due to not
taking duplicate elements.

   s.add("Red"); // s.size() is still 5 because we cannot add duplicate element.
```

```
 s.add("Blue"); // s.size() is still 5 because we cannot add duplicate element.
 System.out.println("Unordered Set Elements");
 System.out.println(s);


// Check 'Black' element is present in the above set or not.
  if(s.equals("Black"))
  {
    System.out.println("Black element is not present in set.");
  }
 else
 {
   s.add("Black");
   System.out.println("Black is added successfully.");
   System.out.println("Set Elements after adding black element");
   System.out.println(s);
  }
// Create another set object to add collection of elements to the above set.
  Set<String> s2 = new LinkedHashSet<String>();
   s2.add("White");
   s2.add("Brown");
```

```
    s2.add("Red"); // Duplicate element.



// Call addAll() method to add all the elements of the given collection.

    s.addAll(s2);

   System.out.println("Set Elements after adding elements from given collection");

   System.out.println(s);

  }

}
```

Output:

```
      Size before adding elements: 0

      Unordered Set Elements

      [Red, Blue, Yellow, Orange, Green]

      Black is added successfully.

      Set Elements after adding black element

      [Red, Blue, Yellow, Black, Orange, Green]

      Set Elements after adding elements from given collection

      [Red, Brown, White, Blue, Yellow, Black, Orange, Green]
```

In the above example program, you can observe in the output, set does not maintain the order of elements while storing. It gives an unordered collection of elements in HashSet.

To impose an order on them, we need to use the LinkedHashSet class, which we will learn in the next section.

When we add duplicate elements in the set, it rejected duplicate elements because the set does not allow duplicate elements (keep in mind these points).

2. Removing an Element from Set: Let's take a program where we will remove an element from set. We will also check the set is empty or not before adding elements in the list.

Program source code 2:

```
package setProgram;

import java.util.HashSet;

import java.util.Set;

public class SetExample2

{

public static void main(String[] args)

{

// Create a generic set object of type String.

   Set<String> s = new HashSet<String>();



// Check set is empty or not.

   boolean check = s.isEmpty(); // Return type of this method is an boolean.

   System.out.println("Is Set empty: " +check);
```

```
// Adding elements to set.
   s.add("Orange");
   s.add("Red");
   s.add("Blue");
   s.add("Yellow");
   s.add("Green");
 if(s.isEmpty())
 {
  System.out.println("Set is empty.");
  }
 else
 {
   System.out.println("Set is not empty.");
   System.out.println("Elements in the Set");
   System.out.println(s);
  }
// Remove an element from set.
   s.remove("Blue");
   System.out.println("Set elements after removing");
   System.out.println(s);
```

```
// Get the total number of set elements.
   int size = s.size();
   System.out.println("Total number of elements: " +size);
  }
}
```

Output:

```
Is Set empty: true
Set is not empty.
Elements in the Set
[Red, Blue, Yellow, Orange, Green]
Set elements after removing
[Red, Yellow, Orange, Green]
Total number of elements: 4
```

3. Searching an Element in Set: Let's make a program where we will search an element in set. The contains() method checks whether an element is present in the set. If it is present in the set, the method returns true, otherwise false.

Program source code 3:

```
package setProgram;

import java.util.HashSet;

import java.util.Set;
```

```
public class SetExample3

{

public static void main(String[] args)

{

 Set<Character> s = new HashSet<Character>();

  s.add('D');

  s.add('F');

  s.add('H');

  s.add('P');

  s.add('K');

  s.add(null);

  s.add(null); // Duplicate null element. Therefore, set allow only one null element.

 System.out.println("Unordered Set Elements");

 System.out.println(s);



// Call contains() method to search an element.

  boolean search = s.contains('A'); // Returns false because A is not present in the set.

System.out.println("Is Element A present in set: " +search);

if(s.contains('K'))

{
```

```
 System.out.println("K is present in set.");

 }

else {

   System.out.println("K is not present in set.");

 }

 int hashcode = s.hashCode();

 System.out.println("HashCode value: " +hashcode);

 }

}
```

Output:

```
      Unordered Set Elements

      [P, null, D, F, H, K]

      Is Element A present in set: false

      K is present in set.

      HashCode value: 365
```

When to use Set?

---

1. If you want to represent a group of individual elements as a single entity where duplicates are not allowed and insertion order is not preserved then we should go for the Set.

2. If your requirement is to get unique elements, set is the best choice for this.

3. If you want to remove duplicate elements without maintaining the insertion order from the non-set collection, you should go for set.

Let's take a simple example program where we will remove duplicate elements from the ArrayList. Look at the source code.

Program source code 4:

```
package setProgram;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

public class SetExample4

{

public static void main(String[] args)

{

// Create a generic list object of type Integer.

  List<Integer> list = new ArrayList<Integer>();

  int size = list.size();

 System.out.println("Size before adding elements: " +size);

  list.add(5);

  list.add(10);
```

```
 list.add(5);

 list.add(15);

 list.add(20);

 list.add(10);

 list.add(20);

 list.add(30);

System.out.println("Original order of List Elements");

System.out.println(list);



Set<Integer> s = new HashSet<Integer>(list);

System.out.println("Unodered List Elements after removing duplicates.");

System.out.println(s);

 }

}
```

Output:

```
    Size before adding elements: 0

    Original order of List Elements

    [5, 10, 5, 15, 20, 10, 20, 30]

    Unodered List Elements after removing duplicates
    [20, 5, 10, 30, 15]
```

Java Collections Notes

Q. Assume that set1 is a set that contains string elements such as Banana, Orange, and Apple. set2 is another set that contains string elements such as Banana, Orange, and Mango.

Answer the following questions:

a) What are the elements of set1 and set2 after executing set1.addAll(set2)?

b) What are the elements of set1 and set2 after executing set1.add(set2)?

c) What are the elements of set1 and set2 after executing set1.removeAll(set2)?

d) What are the elements of set1 and set2 after executing set1.remove(set2)?

e) What are elements of set1 and set2 after executing set1.retainAll(set2)?

f) What is the size of set1 after executing set1.clear()?

g) What is the size of set2 after executing set2.add("Mango")?

Ans a: set1: [Apple, Mango, Orange, Banana], set2: [Mango, Orange, Banana]
Ans b: Compilation error: The method add(String) in the type Set<String> is not applicable for the arguments (Set<String>)
Ans c: set1: [Apple], set2: [Mango, Orange, Banana]
Ans d: set1: [Apple, Orange, Banana], set2: [Mango, Orange, Banana]
Ans e: set1: [Orange, Banana], set2: [Mango, Orange, Banana]
Ans f: After executing set1.clear(), size of set1 is 0.
Ans g: After executing set2.add("Mango"), the size of set2 is 3 because duplicate elements cannot be added to the set.

How to Iterate Set in Java

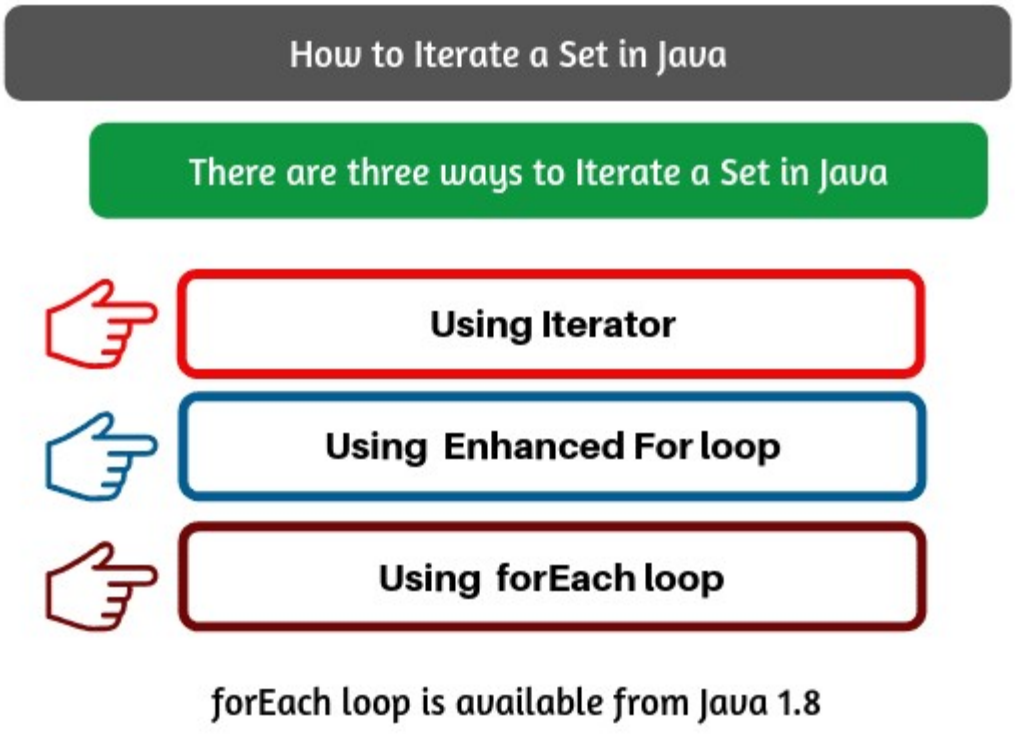
In this tutorial, we will learn **how to iterate Set in Java**. Set interface does not provide any get() method like the *List interface* to retrieve elements.

Therefore, the only way to take out elements from the set can be by using Iterator() method but this method does not return elements from set in any particular order.

Let's see how many ways to iterate set in Java.

There are mainly three ways to iterate a set in Java. We can iterate set by using any one of the following ways. They are as follows:

- Using *Iterator*
- Using Enhanced for loop
- Using forEach()



How to iterate Set using Iterator in Java?

Using Iterator method, we can traverse only in the forward direction from the first to last element. We cannot traverse over elements in the backward direction using the iterator() method.

Java Collections Notes

Let's take an example program where we will iterate elements of set using the iterator() method. Follow all the steps in the coding.

Program source code 1:

```
package iterateSet;

import java.util.HashSet;

import java.util.Iterator;

import java.util.Set;

public class IterateSetEx

{

public static void main(String[] args)

{

// Create a generic set object of type String.

   Set<String> s = new HashSet<String>(); // s.size() is 0.

   int size = s.size();

  System.out.println("Size before adding elements: " +size);



// Adding elements to set.

   s.add("Orange"); // s.size() is 1.

   s.add("Red"); // s.size() is 2.

   s.add("Blue"); // s.size() is 3.
```

```
  s.add("Yellow"); // s.size() is 4.

  s.add("Green"); // s.size() is 5.

System.out.println("Elements in set");

System.out.println(s);



Iterator<String> itr = s.iterator();

System.out.println("Iteration using Iterator method");

while(itr.hasNext())

{

 Object str = itr.next();

 System.out.println(str);

 }

 }

}
```

Output:

```
    Size before adding elements: 0

    Elements in set

    [Red, Blue, Yellow, Orange, Green]

    Iteration using Iterator method

    Red Blue Yellow Orange Green
```

During iteration, we cannot add an element to a set at an iterator position. If we try to add an element during iteration, JVM will throw an exception named **ConcurrentModificationException**.

Key point: Iteration means repeating the same operation multiple times.

Let's take an example program where we will try to add an element during the iteration and see which exception is thrown by JVM?

Program source code 2:

```
import java.util.HashSet;

import java.util.Iterator;

import java.util.Set;



public class AddDemo

{

public static void main(String[] args)

{

 Set<String> set= new HashSet<>();

 set.add("Banana");

 set.add("Orange");

 set.add("Apple");

 set.add("Mango");



 Iterator<String> itr = set.iterator();
```

```
 while(itr.hasNext())

 {

   Object str = itr.next();

   System.out.println(str);

   set.add("Grapes"); // Adding element during iteration. It will throw
ConcurrentModificationException.

  }

}

}

Output:

      Apple

      Exception in thread "main" java.util.ConcurrentModificationException
```

However, we can remove a set element at an iterator position, just as we do with the list iterator.

Let's make a program where we will remove a set element at an iterator position during iteration. Look at the following code to understand better.

Program source code 3:

```
import java.util.HashSet;

import java.util.Iterator;

import java.util.Set;
```

```
public class RemoveDemo

{

public static void main(String[] args)

{

 Set<String> set= new HashSet<>();

 set.add("Banana");

 set.add("Orange");

 set.add("Apple");

 set.add("Mango");



 Iterator<String> itr = set.iterator();

 while(itr.hasNext())

 {

   Object str = itr.next();

   System.out.println(str);



// Removing Mango element.

   if(str.equals("Mango"))

   {

     itr.remove();
```

```
   }

   }

 System.out.println(set);

}

}

Output:

     Apple

     Mango

     Orange

     Banana

     [Apple, Orange, Banana]
```

Key point: List can be iterated by using the ListIterator method but Set cannot be iterated by using ListIterator.

How to iterate Set using Enhanced For loop?

In this section, we will iterate elements of set using enhanced for loop. Let's make a program where we will iterate set elements using enhanced for loop.

Program source code 4:

```
package iterateSet;

import java.util.HashSet;

import java.util.Set;
```

```
public class IterateSetEx2

{

public static void main(String[] args)

{

// Create Set object of type Integer.

   Set<Integer> s = new HashSet<Integer>();

// Adding even numbers betwen 10 to 30 as elements.

   for(int i = 10; i < = 30; i++)

   {

     if(i % 2 == 0)

     {

       s.add(i);

     }

   }

 System.out.println("Even numbers between 10 to 30");

 System.out.println(s);

 System.out.println("Iteration Using Enchanced For Loop");

 for(Integer it:s)

 {

   System.out.println(it);
```

```
  }

 }

}
```

Output:

```
      Even numbers between 10 to 30

      [16, 18, 20, 22, 24, 10, 26, 12, 28, 14, 30]

      Iteration Using Enhanced For Loop

      16 18 20 22 24 10 26 12 28 14 30
```

Iterate Set using forEach loop in Java 1.8?

This is the third way to iterate set elements using for forEach() method. It is a very simple way to iterate elements. Let's take an example program based on this method.

Program source code 5:

```
package iterateSet;

import java.util.HashSet;

import java.util.Set;

public class IterateSetEx3

{

public static void main(String[] args)

{
```

```
Set<Character> s = new HashSet<Character>();

 s.add('A');

 s.add('B');

 s.add('C');

 s.add('D');

 s.add('E');

System.out.println(s);



System.out.println("Iterating using forEach loop in Java 1.8");

 s.forEach(System.out::println);

 }

}
```

Output:

```
[A, B, C, D, E]

Iterating using forEach loop in Java 1.8

A B C D E
```

Key point: forEach() method is available from Java 1.8 onwards.

Java HashSet | Methods, Example

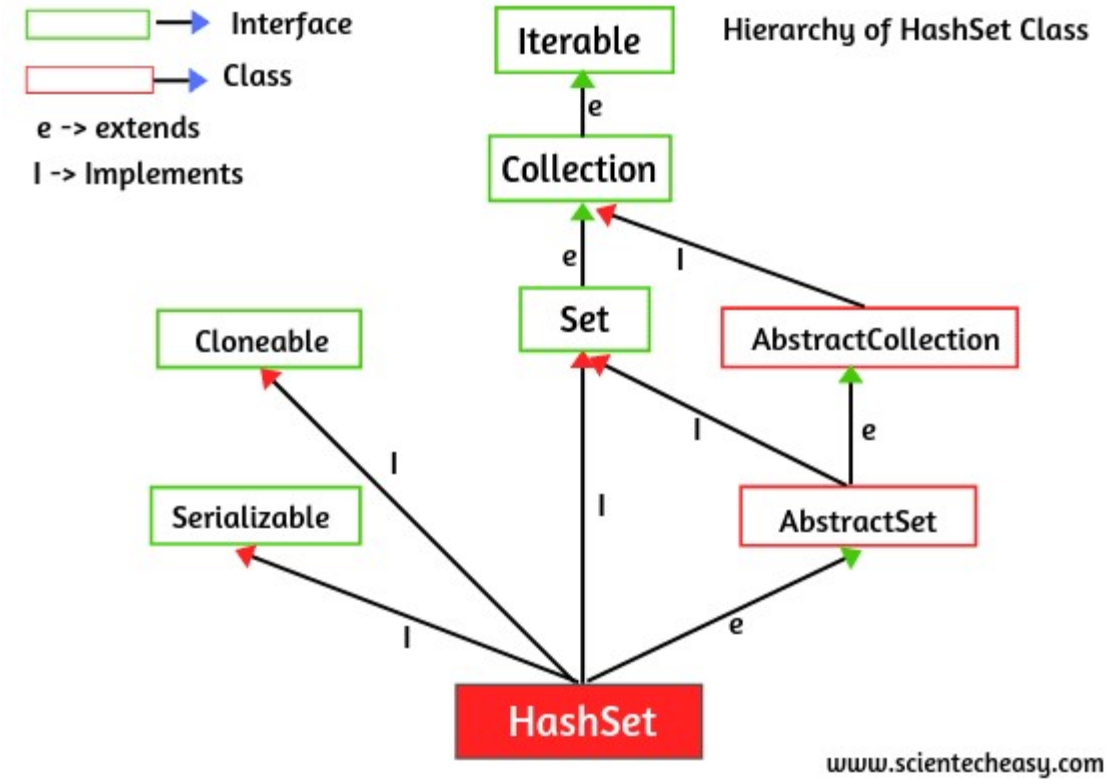
HashSet in Java is an unordered collection of elements (objects) that contains only unique elements. That is, it allows duplicate free elements to be stored.

Java Collections Notes

It internally uses Hashtable data structure to store a set of unique elements. It is much faster and gives constant-time performance for searching and retrieving elements.
HashSet was introduced in Java 1.2 version and it is present in java.util.HashSet package.

Hierarchy of HashSet in Java

HashSet class extends AbstractSet class and implements the Set interface. The AbstractSet class itself extends AbstractCollection class. It also implements cloneable and serializable marker interfaces. The hierarchy diagram of Java HashSet is shown in the below figure.



Java HashSet class declaration

HashSet is a concrete generic class that can be declared in general form as below:

```
public class HashSet<E>

  extends AbstractSet<E>

  implements Set<E>, Cloneable, Serializable
```

Here, E defines the type of elements that the set will hold.

Features of HashSet

There are several important features of Java HashSet that should be kept in mind. They are as follows:

1. The underlying data structure of HashSet is Hashtable. A hash table stores data by using a mechanism called **hashing**.
2. HashSet does not allow duplicate elements. If we try to add a duplicate element in HashSet, the older element would be overwritten.

3. It allows only one null element. If we try to add more than one null element, it would still return only one null value.

4. HashSet does not maintain any order in which elements are added. The order of the elements will be unpredictable. It will return in any random order.

5. It is much faster due to the use of hashing technique and gives constant-time performance for adding (insertion), retrieval, removal, contains, and size operations.

Hashing provides constant execution time for methods like add(), remove(), contains(), and size() even for the large set.

6. HashSet class is not synchronized which means it is not thread-safe. Therefore, multiple threads can use the same HashSet object at the same time and will not give the deterministic final output.

If you want to synchronize HashSet, use Collections.synchronizeSet() method.

7. The iterator returned by HashSet class is fail-fast which means any modification that happened in the HashSet during iteration, will throw ConcurrentModificationException.

Constructors of HashSet class

The following constructors are defined for Java HashSet class. They are as follows:

1. HashSet(): It constructs an empty HashSet (i.e, default HashSet). The default capacity is 16. The syntax to create Hashset object is as follows:

```
HashSet hset = new HashSet();

HashSet<T> hset = new HashSet<T>(); // Generic form. T is the type of object that HashSet will hold.
```

2. HashSet(int initialCapacity): It initializes the capacity of HashSet. When the set capacity reaches full and a new element is added, the capacity of the hash set is expended automatically.
The internal structure will double in size before adding a new element.
The general syntax in generic form is as follows:

```
HashSet<T> hset = new HashSet<T>(int initialCapacity);
```

3. HashSet(int initialCapacity, float fillRatio): This form of constructor initializes capacity and fill ratio (also called load factor or load capacity) of the hash set.

When the number of elements is greater than capacity of HashSet, the size of the HashSet is grown automatically by multiplying capacity with load factor.

The default value of the load factor is 0.75. The value of the fill ratio ranges from 0.0 to1.0. The general syntax to create an object of HashSet with initial capacity and load factor is as follows:

```
HashSet<T> hset = new HashSet<T>(int initialCapacity, float loadFactor);

For example:

  HashSet<Integer> hset = new HashMap<Integer>(30, 0.7f);
```

4. HashSet(Collection c): It initializes HashSet by using elements of c. This constructor acts as a copy constructor. It copies elements from one collection into the newly created collection. We cannot provide a custom initial capacity or fill ratio.
If the original collection had duplicate elements, only one duplicate element will be allowed in the final created set.

Note: Constructors that do not take a load factor, 0.75 is used.

HashSet Class Methods in Java

Java HashSet class does not define any additional methods. It inherits methods of its parent classes and methods of the implemented interface.

The various methods provided by its superclasses and interfaces are as follows:

1. boolean add(Object o): This method is used to add the specified element in this set. It returns true if set does not already contain a specified element.

2. boolean addAll(Collection c): This method is used to add a group of elements from another collection to the set.
Each element in the collection will be added to the current set via calling add() method on each element. It returns true if the current set changes.

If no elements are added, false is returned. A UnsupportedOperationException will be thrown when a current set does not support adding elements.

3. boolean remove(Object o): This method is used to remove the specified element from the set.
To remove an element, set internally use equals() method for each element. If the element is found, element is removed from set and returns true.

If not found, false is returned. If the removal is not supported, you will get UnsupportedOperationException.

4. void clear: This method is used to remove all elements from a set. It returns nothing.
5. boolean contains(Object element): To check a specific element is present or not in the set, use contains() method.
If the specified element is found in the set, it returns true otherwise false. The equality checking is done through the element's equal method.

6. int size(): If you wish to know how many elements are in a set, call size() method.
7. boolean isEmpty(): If you wish to check whether HashSet contains elements or not. Call isEmpty() method.
8. Object clone(): Since HashSet implements cloneable interface. So, we can create a shallow copy of that hash set by calling clone() method of HashSet.
9. boolean equals(Object o): The HashSet class defines equality by using equals() method on each element within the set.

10. int hashCode(): The HashSet class overrides the hashCode() method to define an appropriate hash code value for the set. It returns the same hash code by sum up all hash codes of all the elements.

Java HashSet Example Program

---

1. Let's take an example program where we will add elements to set. We will also add duplicate elements but HashSet will not store them second time because hash set does not allow to add duplicate data.

In this example program, we will add null element into the set. Look at the program source code below.

Program source code 1:

```
package hashSetTest;

import java.util.HashSet;

public class HashSetExample1 {

public static void main(String[] args)

{

// Create a HashSet object.

   HashSet<String> set = new HashSet<String>(); // An empty hash set.



// Adding elements to HashSet.

   set.add("First");

   set.add("Second");
```

```
  set.add("Third");

  set.add("Fourth");

  set.add("Fifth");



// Adding duplicate elements that will be ignored.

  set.add("First");

  set.add("Third");



// Adding of null elements.

  set.add(null);

  set.add(null); // Ignored.

 System.out.println("Unordered and No Duplicate HashSet Elements");

 System.out.println(set);

 }

}

Output:

      Unordered and No Duplicate HashSet Elements

      [null, Second, Third, First, Fourth, Fifth]
```

As you can observe in the output that the set does not maintain the same order of elements in which they have been inserted.

2. Let's take an example program where we will see how to add elements from the existing collection to HashSet in Java.

Program source code 2:

```
package hashSetTest;

import java.util.ArrayList;

import java.util.HashSet;

public class HashSetExample2 {

public static void main(String[] args)

{

// Create an ArrayList object.

   ArrayList<String> al = new ArrayList<String>();

    al.add("Monday");

    al.add("Tuesday");

    al.add("Wednesday");

    al.add("Thursday");

    al.add("Friday");



// Adding duplicate elements.

    al.add("Monday");

    al.add("Friday");

   System.out.println("Original Elements Order ");
```

```
  System.out.println(al);



// Create HashSet object.

  HashSet<String> hset = new HashSet<String>();



// Call addAll() method for adding all elements from existing collection to HashSet.

  hset.addAll(al);

 System.out.println("Unordered HashSet Elements without Duplicate elements");

 System.out.println(hset);

 }

}

Output:

     Original Elements Order

     [Monday, Tuesday, Wednesday, Thursday, Friday, Monday, Friday]

     Unordered HashSet Elements without Duplicate elements

     [Monday, Thursday, Friday, Wednesday, Tuesday]
```

3. Let's take an example program where we will see how to remove a specific element from a HashSet and remove all elements available in a set.

Program source code 3:

```
package hashSetTest;
```

```
import java.util.HashSet;

public class HashSetExample3 {

public static void main(String[] args)

{

 HashSet<Integer> hset = new HashSet<Integer>();

  hset.add(5);

  hset.add(10);

  hset.add(15);

  hset.add(20);

System.out.println("Initial list of elements");

System.out.println(hset);



// Removing a specific element from HashSet.

   hset.remove(10);

   System.out.println("List of elements after removing 10");

   System.out.println(hset);



 HashSet<Integer> hset2 = new HashSet<Integer>();

  hset2.add(10);

  hset2.add(25);
```

```
 hset.addAll(hset2);
 System.out.println("List of Elements after adding elements from existing collection");
 System.out.println(hset);


// Removing all new elements from HashSet.
  hset.removeAll(hset2);
  System.out.println("List of Elements after removing elements from hset2");
  System.out.println(hset);


// Removing all elements available in HashSet.
  clear();
  System.out.println("After invoking clear() method: "+hset);
 }
}
Output:
    Initial list of elements
    [20, 5, 10, 15]
    List of elements after removing 10
    [20, 5, 15]
    List of Elements after adding elements from existing collection
```

```
[20, 5, 25, 10, 15]

List of Elements after removing elements from hset2

[20, 5, 15]

After invoking clear() method: [ ]
```

4. Let's create a program where we will check the number of elements in Java HashSet. We will also verify that HashSet is empty or not. Look at the source code to understand better.

Program source code 4:

```
package hashSetTest;

import java.util.HashSet;

import java.util.Set;

public class HashSetExample4 {

public static void main(String[] args)

{

  Set<String> pCountry = new HashSet<String>();



// Check that HashSet is empty or not.

  System.out.println("Is popularCountries set empty? : " + pCountry.isEmpty());

  System.out.println("Number of countries in HashSet before adding: "
+pCountry.size());

   pCountry.add("INDIA");

   pCountry.add("USA");
```

```
    pCountry.add("UK");

    pCountry.add("FRANCE");



// Find size of HashSet.

   System.out.println("Number of countries in HashSet after adding: " + pCountry.size());

  }

}

Output:

     Is popularCountries set empty? : true

     Number of countries in HashSet before adding: 0

     Number of countries in HashSet after adding: 4
```

5. Let's create a program where we will check the existing element in HashSet. We will use contains() method for this.

Program source code 5:

```
package hashSetTest;

import java.util.HashSet;

import java.util.Set;

public class HashSetExample4 {

public static void main(String[] args)

{

 Set<String> set = new HashSet<String>();
```

```
System.out.println("Is set empty? : " + set.isEmpty());

System.out.println("Number of elements in HashSet before adding: " +set.size());



 set.add("Dollar");

 set.add("Indian Rupee");

 set.add("Euro");

 set.add("Yen");



System.out.println("List of Elements in set");

System.out.println(set);

System.out.println("Number of elements in the HashSet after adding: " + set.size());



// Call contains() method to check an element exists in set or not.

  if(set.contains("Dollar")){

    System.out.println("Does Element 'Dollar' exist in set?");

    System.out.println("Yes, Element 'Dollar' exists in set");

  }

else{

  System.out.println("No, Element 'Dollar' does not exist in set");

 }
```

```
System.out.println("Does Element 'Dinar' exist in set?");

if(set.contains("Dinar")){

  System.out.println("Yes, Element 'Dinar' exists in set ");

 }

else {

   System.out.println("No, Element 'Dinar' does not exist in set");

  }

 }

}
```

Output:

```
    Is set empty? : true

    Number of elements in HashSet before adding: 0

    List of Elements in set

    [Yen, Dollar, Indian Rupee, Euro]

    Number of elements in HashSet after adding: 4

    Does Element 'Dollar' exist in set?

    Yes, Element 'Dollar' exists in set

    Does Element 'Dinar' exist in the set?

    No, Element 'Dinar' does not exist in the set
```

How to create User-defined Object of HashSet?

Let's create a program where we will create a user-defined object of HashSet in Java. Look at the source code to understand better.

Program source code 6:

```
package hashSetTest;

public class Student

{

// Declare instance variables.

   String name, sName;

   int id;

public Student(String name, String sName, int id)

{

  this.name = name;

  this.sName = sName;

  this.id = id;

  }

 }

package hashSetTest;

import java.util.HashSet;

public class HashSetExample6

{
```

```
public static void main(String[] args)

 {

// Create a user-defined HashSet object of type Student.

   HashSet<Student> hset = new HashSet<Student>();



// Create objects of Student class and pass the parameters to their constructors.

   Student s1 = new Student("John", "RSVM", 0012);

   Student s2 = new Student("Shubh", "DPS", 1234);

   Student s3 = new Student("Ricky", "DAV", 9876);



// Adding elements to HashSet and pass reference variables s1, s2, s3.

    hset.add(s1);

    hset.add(s2);

    hset.add(s3);

// Traversing HashSet.

   for(Student s:hset)

   {

     System.out.println(s.name+" "+s.sName+" "+s.id);

    }

 }
```

```
}

Output:

    Ricky DAV 9876

    John RSVM 10

    Shubh DPS 1234
```

When to use HashSet in Java?

HashSet in Java is used when

1. We don't want to store duplicate elements.
2. We want to remove duplicate elements from the list.
3. HashSet is more preferred when add and remove operations are more as compared to get operations.
4. We are not working in a multithreading environment.

LinkedHashSet in Java | Example Program

LinkedHashSet in Java is a concrete class that implements set interface and extends HashSet class with a doubly linked list implementation.
It internally uses a linked list to store the elements in the set. It was added in Java 1.4 version.
Java LinkedHashSet class is the same as HashSet class, except that it maintains the ordering of elements in the set in which they are inserted.

That is, LinkedHashSet not only uses a hash table for storing elements but also maintains a double-linked list of the elements in the order during iteration.

In simple words, elements in the HashSet are not ordered, but elements in the LinkedHashSet can be retrieved in the same order in which they were inserted into the set.

Hierarchy of LinkedHashSet class in Java

The hierarchy diagram of Java LinkedHashSet is shown in the below figure.

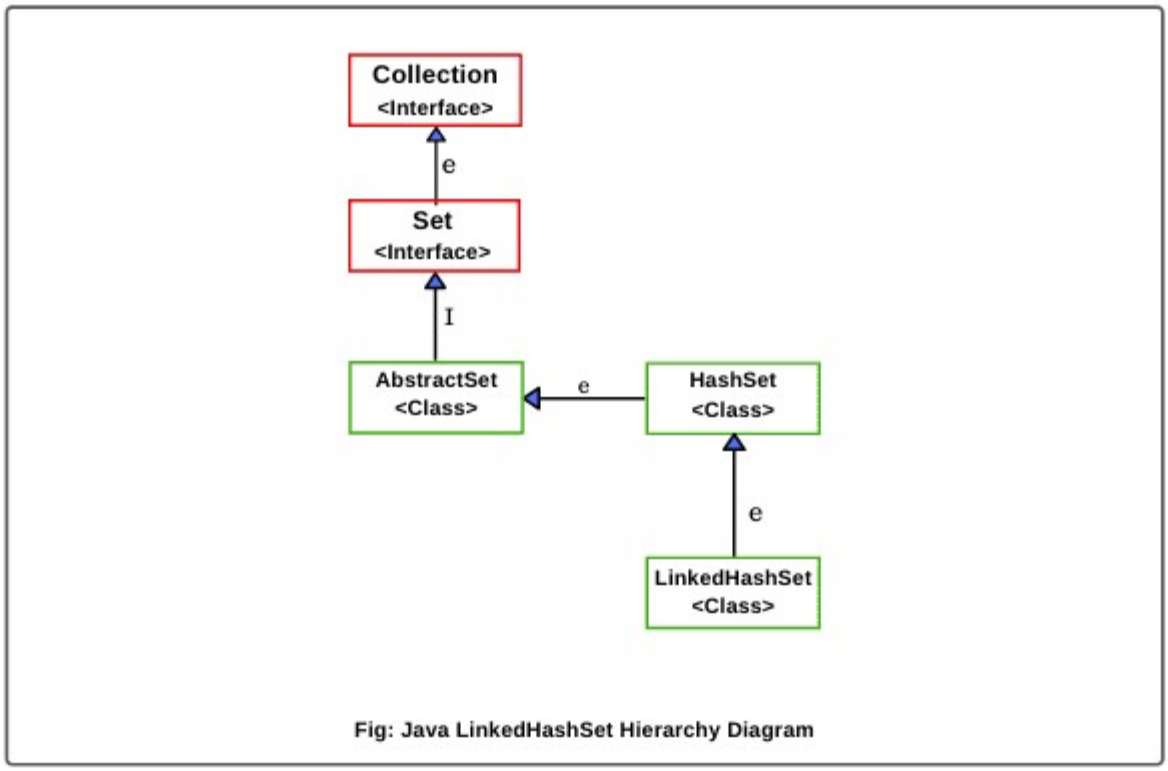


Fig: Java LinkedHashSet Hierarchy Diagram

LinkedHashSet class does not define any extra methods of its own. Since LinkedHashSet class extends HashSet and implements the Set interface,

therefore, all the methods defined by the Set interface and HashSet class can be used while using LinkedHashSet class.

Java LinkedHashSet Class Declaration

LinkedHashSet is a generic class that has declaration given below:

```
class LinkedHashSet<T>
```

Here, T defines the generic type parameter that represents the data type of elements that the LinkedHashSet will hold.

Features of LinkedHashSet

The important features of Java LinkedHashSet class are as follows that needs to keep in mind.

1. Java LinkedHashSet class contains unique elements like HashSet. It does not allow to insert of duplicate elements. If we try to add a duplicate element, it will fail and the iteration order of the set is not modified.

2. LinkedHashSet class permits to insert null element.

3. LinkedHashSet class in Java is non-synchronized. That means it is not thread-safe.

4. LinkedHashSet class preserve the insertion order of elements

5. It is slightly slower than HashSet.

6. Linked hash set is very efficient for insertion and deletion of elements.

Constructor of LinkedHashSet in Java

We can create LinkedHashSet by using one of its four constructors. They are as follows:

1. LinkedHashSet( ): This constructor is used to create an empty LinkedHashSet. It is a default LinkedHashSet. The general syntax to create LinkedHashSet is as follows:

```
LinkedHashSet<T> lhset = new LinkedHashSet<T>();
```

2. LinkedHashSet(Collection c): This constructor is used to initialize the LinkedHashSet with elements of collection c. The general syntax is as follow:

```
LinkedHashSet<T> lhset = new LinkedHashSet<T>(Collection c);
```

3. LinkedHashSet(int initialCapacity): This constructor is used to create LinkedHashSet with initializing the capacity of the linked hash set. It takes an integer value to initialize capacity. The general syntax to create linked hash set in Java is given below:

```
LinkedHashSet<T> lhset = new LinkedHashSet<T>(int size);
```

4. LinkedHashSet(int initialCapacity, float loadFactor): This constructor is used to create LinkedHashSet with initializing both the capacity and load factor of the linked hash set.

```
LinkedHashSet<T> lhset = new LinkedHashSet<T>(int initialCapacity, float loadFactor)
```

These constructors work the same as the constructors for HashSet.

Java LinkedHashSet Example Programs

In this section, we will take some example program where we will perform a few frequently used operations on the Java LinkedHashSet.

1. **Adding elements:** Let's create a program where we will perform operations such as adding, checking the size of LinkedHashSet, etc. Look at the program source code to understand better.

Program source code 1:

```
import java.util.LinkedHashSet;

public class AddTest

{

public static void main(String[] args)

{

// Create a Linked hash set of generic type.

 LinkedHashSet<String> lhset= new LinkedHashSet<String>();



// Checking the size of LinkedHashSet before adding elements.

  int size = lhset.size();

  System.out.println("Size of LinkedHashSet before adding elements: " +size);



// Adding elements in the linked hash set.

   lhset.add("Red"); // lhset.size() is 1.

   lhset.add("Green"); // lhset.size() is 2.

   lhset.add("Yellow"); // lhset.size() is 3.

   lhset.add("Blue"); // lhset.size() is 4.

   lhset.add("Orange"); // lhset.size() is 5.
```

```
System.out.println("Elements in Set: " +lhset);

int size2 = lhset.size();

System.out.println("Size of LinkedHashSet after adding elements: " +size2);



// Adding duplicate elements that already exist in set.

  lhset.add("Red"); // lhset.size() is still 5.

  lhset.add("Yellow"); // lhset.size() is still 5.



// Create another set of String type.

   LinkedHashSet<String> lhset2 = new LinkedHashSet<String>();

   lhset2.add("Brown");

   lhset2.add(null);



// Adding elements of set2 into set.

   lhset.addAll(lhset2);

   System.out.println("Elements in Set after adding: " +lhset);

 }

}
```

Output:

```
    Size of LinkedHashSet before adding elements: 0
```

```
Elements in Set: [Red, Green, Yellow, Blue, Orange]

Size of LinkedHashSet after adding elements: 5

Elements in Set after adding: [Red, Green, Yellow, Blue, Orange, Brown, null]
```

2. **Removing element:** Let's create another program where we will remove an element from the linked hash set.

Program source code 2:

```
import java.util.LinkedHashSet;

public class RemoveDemo

{

public static void main(String[] args)

{

// Create a Linked hash set of generic type.

 LinkedHashSet<String> set= new LinkedHashSet<String>();



// Adding elements in the linked hash set.

   set.add("A");

   set.add("G");

   set.add("Y");

   set.add("B");

   set.add("O");

   set.add(null);
```

```
System.out.println("Elements in set: " +set);


// Remove a string element from linked hash set.
   set.remove(null);
   System.out.println("Elements in set after removing: " +set);
   System.out.println(set.size()+ " elements in set");


// Create another linked hash set of String type.
   LinkedHashSet<String> set2 = new LinkedHashSet<String>();
   set2.add("S");
   set2.add(null);
   System.out.println("Elements in set2: " +set2);
   System.out.println(set2.size()+ " elements in set2");


  System.out.println("Is S in set2? " +set2.contains("S"));


  set.addAll(set2);
  System.out.println("Elements in set after adding: " +set);
```

```
  set.removeAll(set2);

  System.out.println("Elements in set after removing set2: " +set);



  set.retainAll(set2);

  System.out.println("After removing common elements in set2 " + "from set, set is " +
set);

  }

}
```

Output:

```
    Elements in set after removing: [A, G, Y, B, O]

    5 elements in set

    Elements in set2: [S, null]

    2 elements in set2

    Is S in set2? true

    Elements in set after adding: [A, G, Y, B, O, S, null]

    Elements in set after removing set2: [A, G, Y, B, O]

    After removing common elements in set2 from set, set is []
```

3. **Removing duplicate elements:** Let's make a program where we will remove duplicate numbers from ArrayList using LinkedHashSet.

Program source code 3:

```
import java.util.ArrayList;

import java.util.LinkedHashSet;
```

```
public class RemovingDuplicate {

public static void main(String[] args)

{

int[] num = {20, 30, 50, 30, 40, 80, 10, 10};

ArrayList<Integer> ar = new ArrayList<Integer>();



// Adding numbers to the array list.

  for(int i = 0; i < num.length; i++) {

                ar.add(num[i]);

  }

 System.out.println("Original list: " +ar);

 LinkedHashSet<Integer> lhset = new LinkedHashSet<>(ar);

 System.out.println("New list after removing dupliacte numbers: " +lhset);

 }

}
```

Output:

Original list: [20, 30, 50, 30, 40, 80, 10, 10]

New list after removing dupliacte numbers: [20, 30, 50, 40, 80, 10]

As you can observe in the output, after removing duplicate elements from ArrayList, LinkedHashSet maintains the insertion order of elements in which they had been inserted into the list.

4. **Iterating LinkedHashSet:** Let's take an example program where we will iterate elements of LinkedHashSet using iterator() method and enhanced for loop. Look at the following source code.

Program source code 4:

```
import java.util.Iterator;

import java.util.LinkedHashSet;

public class IteratingLinkedHashSet {

public static void main(String[] args)

{

LinkedHashSet<String> lhset = new LinkedHashSet<>();

  lhset.add("New York");

  lhset.add("Dhanbad");

  lhset.add("Sydney");

  lhset.add("Cape Town");

  lhset.add("London");



// Iterating elements of LinkedHashSet using iterator() method.

 System.out.println("Iteration using iterator");

  Iterator<String> itr = lhset.iterator();
```

```
 while(itr.hasNext())

 {

 System.out.println(itr.next());

 }

System.out.println();

// Iterating elements of LinkedHashSet using enhanced for loop

System.out.println("Iteration using enhanced for loop");

 for (String s : lhset)

   System.out.print(s + " ");

   System.out.println();

 }

}

Output:

    Iteration using iterator

    New York

    Dhanbad

    Sydney

    Cape Town

    London
```

```
Iteration using enhanced for loop

New York Dhanbad Sydney Cape Town London
```

5. **Adding custom objects:** Let's take an example program where we will add custom objects of type Student into LinkedHashSet and iterate it. Look at the following source code.

Program source code 5:

```
public class Student {

String name;

int id;

String city;



Student(String name, int id, String city){

 this.name = name;

 this.id = id;

 this.city = city;

  }

}

import java.util.LinkedHashSet;

public class StudentInfo {

public static void main(String[] args)

{
```

```
LinkedHashSet<Student> lhset = new LinkedHashSet<Student>();



// Creating objects of Students.

   Student st1 = new Student("John", 2345, "New York");

   Student st2 = new Student("Deep", 1234, "Dhanbad");

   Student st3 = new Student("Ricky", 7583, "Cape Town");



// Adding elements (object references) into LinkedHashSet.

   lhset.add(st1);

   lhset.add(st2);

   lhset.add(st3);



// Traversing linked hash set.

  for(Student s:lhset){

      System.out.println("Name: " +s.name+" "+ "Id: " +s.id+" "+"City: "+s.city);

    }

  }

}

Output:

     Name: John Id: 2345 City: New York
```

Java Collections Notes

```
Name: Deep Id: 1234 City: Dhanbad

Name: Ricky Id: 7583 City: Cape Town
```

When to use LinkedHashSet in Java?

LinkedHashSet can be used when you do not want duplicate elements (i.e. want to remove duplicate elements) and want to maintain order in which elements are inserted.

If you want to impose different orders such as increasing or decreasing order, you can use TreeSet class that you will learn in the next tutorial.

Which is better to use: HashSet or LinkedHashSet?

If you do not require to maintain order in which elements are inserted then use HashSet that is more fast and efficient than LinkedHashSet.

TreeSet in Java | Methods, Example

A **TreeSet in Java** is another important implementation of the Set interface that is similar to the HashSet class, with one added improvement.
It sorts elements in ascending order while HashSet does not maintain any order.

Java TreeSet implements SortedSet interface. It is a collection for storing a set of unique elements (objects) according to their natural ordering.

It creates a sorted collection that uses a tree structure for the storage of elements or objects. In simple words, elements are kept in sorted, ascending order in the tree set.

For example, a set of books might be kept by height or alphabetically by title and author.

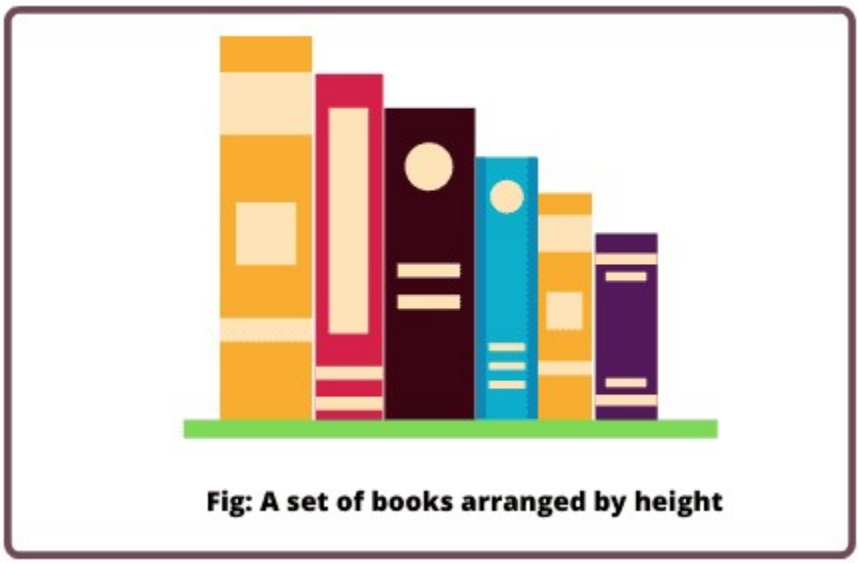


Fig: A set of books arranged by height

In Java TreeSet, access and retrieval of elements are quite fast because of using tree structure. Therefore, TreeSet is an excellent choice for quick and fast access to large amounts of sorted data.

The only restriction with using tree set is that we cannot add duplicate elements in the tree set.

TreeSet class declaration in Java

TreeSet is a generic class that is declared as:

```
class TreeSet<T>
```

In this syntax, T defines the type of objects or elements that the set will hold.

Hierarchy of TreeSet class in Java

TreeSet class extends AbstractSet and implements NavigableSet interface. NavigableSet extends SortedSet to provide navigation methods.

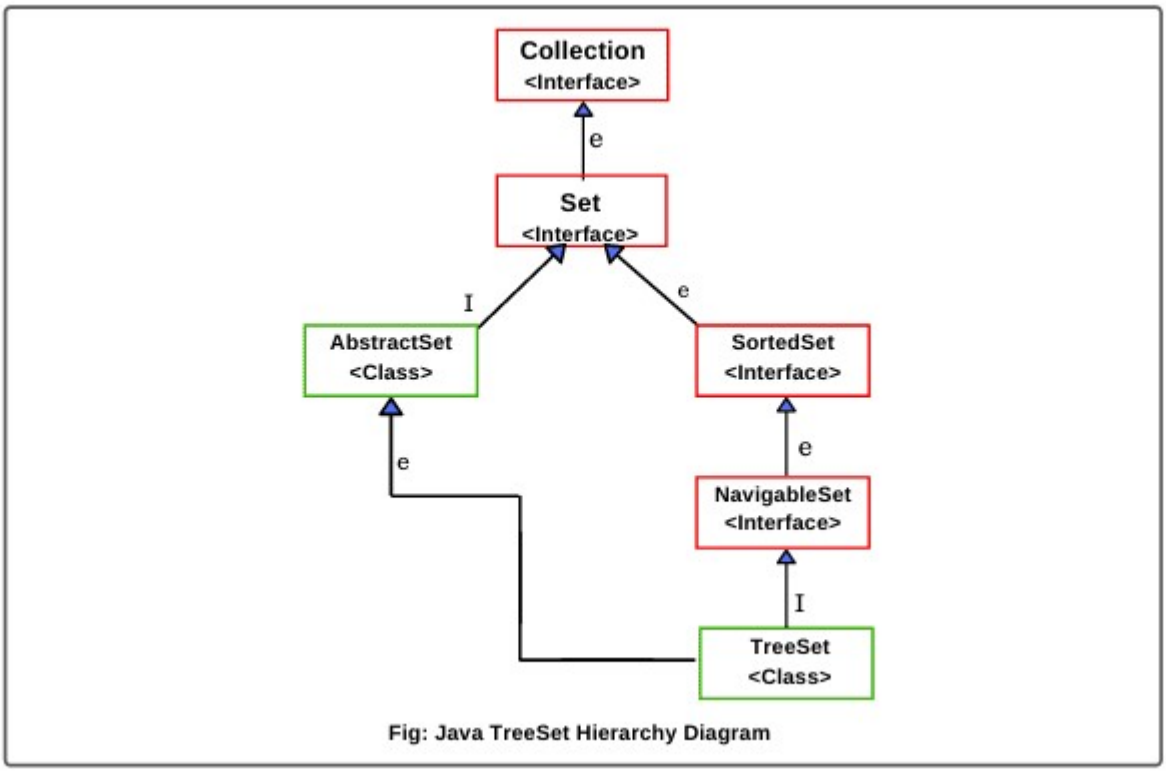


Fig: Java TreeSet Hierarchy Diagram

Features of TreeSet class in Java

There are several important features of TreeSet class in java that must be kept in mind. They are as:

1. Java TreeSet contains unique elements similar to the HashSet. It does not allow the addition of a duplicate element.

2. The access and retrieval times are quite fast.

3. TreeSet does not allow inserting null element.

4. TreeSet class is non-synchronized. That means it is not thread-safe.

5. TreeSet maintains the ascending order. When we add elements into the collection in any order, the values are automatically presented in sorted, ascending order.

6. Java TreeSet internally uses a TreeMap for storing elements.

How to create TreeSet in Java?

TreeSet has the following constructors. We can create a TreeSet instance by using one of its four constructors.

1. TreeSet( ): This default constructor creates an empty TreeSet that will be sorted in ascending order according to the natural order of its elements.
2. TreeSet(Collection c): It creates a tree set and adds the elements from a collection c according to the natural ordering of its elements. All the elements added into the new set must implement Comparable interface.
If collection c's elements do not implement Comparable interface or are not mutually comparable, this constructor throws ClassCastException.

If collection c contains null reference, this constructor throws NullPointerException.

3. TreeSet(Comparator comp): This constructor creates an empty tree set that will be sorted according to the comparator specified by comp. All elements in the tree set are compared mutually by the specified comparator. A comparator is an interface that is used to implement the ordering of data.
4. TreeSet(SortedSet s): This constructor creates a tree set that contains the elements of sorted set s.

Important TreeSet Methods in Java

Java Collections Notes

TreeSet class in java provides a variety of methods to perform different tasks. They are as follows:

1. boolean add(Object o): This method is used to add the specified element to this set if it is not already present.
2. boolean addAll(Collection c): This method is used to add all the elements of the specified collection to the set.
3. void clear(): It is used to remove all the elements from the set.
4. boolean isEmpty(): This method is used to check that the set has elements or not. It returns true if the set contains no elements otherwise returns false.
5. boolean remove(Object o): This method is used to remove the specified element from the set if it is present.
6. boolean contains(Object o): It returns true if the set has the specified element otherwise returns false.
7. int size(): This method is used to get the total number of elements in the set.
8. Iterator iterator(): The iterator() method is used to iterate the elements in ascending order.

9. Spliterator spliterator(): The splititerator() method is used to create a late-binding and fail-fast spliterator over the elements in the tree set.

10. Object clone(): The clone() method is used to get a shallow copy of this TreeSet instance.
11. Iterator descendingIterator(): It is used to iterate the elements in descending order.

Important Methods Defined by SortedSet Interface

Since Java TreeSet implements SortedSet interface, all the methods defined by SortedSet interface can be used while using TreeSet class. They are as:

1. Object first(): It is used to get the first (lowest) element currently in the sorted set.
2. Object last(): This method returns the last (highest) element currently in the sorted set.
3. Comparator comparator(): It returns comparator that is used to order elements in the set. If the TreeSet uses the natural ordering, this method returns null.
4. SortedSet headSet(Object toObject): This method returns the collection of elements that are less than the specified element.
5. SortedSet subSet(Object fromElement, Object toElement): It returns elements from the set that lie between the given range in which fromElement is included and toElement is excluded.
6. SortedSet tailSet(Object fromElement): It returns elements from the set that is greater than or equal to the specified element.

Important Methods Defined by NavigableSet Interface

Since Java TreeSet Class implements NavigableSet interface, therefore, all methods of this interface can be used while using tree set class. They are as:

1. Object ceiling(Object o): It returns the lowest element from the set equal to or greater than the specified element. If no such element is found, it returns null.
2. Object floor(Object o): It returns the greatest element from the set equal to or less than the specified element. If no such element is found, it returns null element.
3. Object lower(Object o): This method returns the largest element from the set strictly less than the specified element. If no such element is found, it will return null element.

4. Object higher(Object o): This method returns the smallest element from the set strictly greater than the specified element. If no such element is found, it will return null element.
5. Object pollFirst(): It is used to remove and retrieve the first element in the tree set.
6. Object pollLast(): It is used to remove and retrieve the last element in the tree set.
7. NavigableSet descendingSet(): This method returns elements in reverse order.
8. NavigableSet headSet(Object toObject, boolean inclusive): This method returns the collection of elements that are less than or equal to (if, inclusive is true) the specified element.
9. NavigableSet subSet(Object fromElement, boolean fromInclusive, Object toElement, boolean toInclusive): This method returns elements from the set that lie between the specified range.
10. NavigableSet tailSet(Object fromElement, boolean inclusive): It returns elements from the set that is greater than or equal to (if, inclusive is true) the specified element.

Java TreeSet Example Programs

Let's take different types of example programs based on the above methods defined by TreeSet, SortedSet, and NavigableSet.

Let's create a program where we will perform different operations such as adding, checking the size of TreeSet, and the set is empty or not. Look at the source code to understand better.

Program source code 1:

```
import java.util.Set;

import java.util.TreeSet;

public class TreeSetEx1 {
```

```
public static void main(String[] args)

{

// Create a tree set.

 Set<String> ts = new TreeSet<>();



// Check Set is empty or not.

  boolean empty = ts.isEmpty();

  System.out.println("Is TreeSet empty: " +empty);



// Checking the size of TreeSet before adding elements into it.

  int size = ts.size();

  System.out.println("Size of TreeSet: " +size);



// Adding elements into TreeSet.

   ts.add("India"); // ts.size() is 1.

   ts.add("USA"); // ts.size() is 2.

   ts.add("Australia"); // ts.size() is 3.

   ts.add("New zealand"); // ts.size() is 4.

   ts.add("Switzerland"); // ts.size() is 5.
```

```
System.out.println("Sorted TreeSet: " +ts);

int size2 = ts.size();

System.out.println("Size of TreeSet after adding elements: " +size2);

  }

}
```

Output:

```
    Is TreeSet empty: true

    Size of TreeSet: 0

    Sorted TreeSet: [Australia, India, New zealand, Switzerland, USA]

    Size of TreeSet after adding elements: 5
```

Let's make a program where we will perform operations like removing an element and checking of a specific element.

Program source code 2:

```
import java.util.TreeSet;

public class TreeSetEx2

{

public static void main(String[] args)

{

 TreeSet<String> ts = new TreeSet<>();



// Add Strings to tree set.
```

```
  ts.add("India");
  ts.add("USA");
  ts.add("Australia");
  ts.add("New zealand");
  ts.add("Switzerland");

// Checking for a specific element in set.
  boolean element = ts.contains("USA");
  System.out.println("Is USA in TreeSet: " +element);

// Removing element from the tree set.
   ts.remove("New zealand");
   System.out.println("Sorted tree set: " +ts);
   ts.clear();
   System.out.println("Elements in tree set: " +ts);
  }
}
Output:
      Is USA in TreeSet: true
      Elements in tree set: [Australia, India, Switzerland, USA]
```

```
Elements in tree set: []
```

Now we will make a program to perform different operations based on methods defined by the SortedSet interface. Look at the source code.

Program source code 3:

```
import java.util.HashSet;

import java.util.Set;

import java.util.SortedSet;

import java.util.TreeSet;

public class TreeSetEx3

{

public static void main(String[] args)

{

 Set<String> s = new HashSet<>();

  s.add("Delhi");

  s.add("New York");

  s.add("Paris");

  s.add("London");

  s.add("Delhi"); // Adding duplicate elements.



TreeSet<String> ts = new TreeSet<>(s);

System.out.println("Sorted TreeSet: " +ts);
```

```
// Using methods of SortedSet interface.
  System.out.println("First Element: " +ts.first());
  System.out.println("Last Element: " +ts.last());
  System.out.println("HeadSet Elements: " +ts.headSet("London"));
  System.out.println("TailSet Elements: " +ts.tailSet("London"));

SortedSet<String> subSet = ts.subSet("Delhi", "Paris");
System.out.println("SubSet Elements: " +subSet);
System.out.println("Sorted Set: " +ts.comparator()); // It will return null because natural ordering is used.
  }
}
```

Output:

```
    Sorted TreeSet: [Delhi, London, New York, Paris]
    First Element: Delhi
    Last Element: Paris
    HeadSet Elements: [Delhi]
    TailSet Elements: [London, New York, Paris]
    SubSet Elements: [Delhi, London, New York]
```

```
Sorted Set: null
```

Let's take an example program where we will perform operations based on NavigableSet interface methods.

Program source code 4:

```
import java.util.TreeSet;

public class TreeSetEx4

{

public static void main(String[] args)

{

TreeSet<Integer> ts = new TreeSet<>();

 ts.add(25);

 ts.add(80);

 ts.add(05);

 ts.add(100);

 ts.add(90);

 ts.add(200);

 ts.add(300);

System.out.println("Sorted TreeSet: " +ts);



// Using methods of NavigableSet interface.

  System.out.println("Largest element less than 100: " +ts.lower(100));
```

```
System.out.println("Smallest element greater than 100: " +ts.higher(100));

System.out.println("Floor: " +ts.floor(85));

System.out.println("Ceiling: " +ts.ceiling(10));



System.out.println(ts.pollFirst()); // Remove and retrieve the first element from the set.

System.out.println(ts.pollLast()); // Remove and retrieve the last element from the set.

System.out.println("New Treeset: " +ts);



System.out.println("HeadSet: " +ts.headSet(90,true));

System.out.println("SubSet: " +ts.subSet(90, true, 200, true));

}

}
```

Output:

```
    Sorted TreeSet: [5, 25, 80, 90, 100, 200, 300]

    Largest element less than 100: 90

    Smallest element greater than 100: 200

    Floor: 80

    Ceiling: 25

    5

    300
```

```
New Treeset: [25, 80, 90, 100, 200]

HeadSet: [25, 80, 90]

SubSet: [90, 100, 200]
```

How to iterate TreeSet in Java?

In this section, we will iterate elements of TreeSet using iterator() method in ascending and descending order. Look at the following source code.

Program source code 5:

```
import java.util.Iterator;

import java.util.TreeSet;

public class KeySetDemo

{

public static void main(String[] args)

{

TreeSet<Integer> ts = new TreeSet<>();

 ts.add(25);

 ts.add(80);

 ts.add(05);

 ts.add(100);

 ts.add(90);
```

```
System.out.println("Sorted TreeSet:");
// Traversing elements.
 Iterator<Integer> itr = ts.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
}
System.out.println("Iterating elements through Iterator in descending order");
Iterator<Integer> it = ts.descendingIterator();
while(it.hasNext())
{
  System.out.println(it.next());
 }
}
}
```

Output:

```
Sorted TreeSet:
5
25
80
```

```
90

100

Iterating elements through Iterator in descending order

100

90

80

25

5
```

How to sort TreeSet in Java | Ordering of Elements in TreeSet

---

A TreeSet in Java determines the order of elements in either of two ways:

1. A tree set sorts the natural ordering of elements when it implements java.lang.Comparable interface. The ordering produced by comparable interface is called natural ordering. The syntax is given below:

```
public interface Comparable {

public int compareTo(Object o); // Abstract method.



}
```

The compareTo() method of this interface is implemented by TreeSet class to compare the current element with element passed in as a parameter for the order.

If the element argument is less than the current element, the method returns +ve integer, zero if they are equal, or a -ve integer if element argument is greater.

2. TreeSet in Java also determines the order of elements by implementing the Comparator interface. This technique is used when TreeSet class needs to impose a different sorting algorithm regardless of the natural ordering of elements.

The comparator interface provides two methods in which compare() method is more important. The syntax is as follows:

```
public interface Comparator {

public int Compare(Element e1, Element e2); // Abstract method.

public boolean equals(Element e); // Abstract methods.



}
```

The compare() method of this interface accepts two objects (elements) arguments and returns an integer value that specifies their sort order.

This method returns a -ve value when the first element is less than second element, zero if they are equal, or +ve value if the first element is greater.

When to Use TreeSet in Java?

TreeSet can be used when we want unique elements in sorted order.

Java Collections Notes

Which is better to use: HashSet or TreeSet?

If you want to store unique elements in sorted order then use TreeSet, otherwise, use HashSet with no ordering of elements. This is because HashSet is much faster than TreeSet.